



# Overview of Computer Science

Phillip Barry

# Overview of Computer Science

Phillip Barry

Department of Computer Science and Engineering

University of Minnesota-Twin Cities

Copyright ©2015 Phillip Barry

This textbook consists of notes for the CSci 1001 Overview of Computer Science class at the University of Minnesota-Twin Cities. More information about that class and these notes are in the opening chapter. The original version of these notes was used in the Spring 2014 offering of that class. This current version, Version 1.2, is a small update, containing some minor clarifications and corrections, as well as the addition of a chapter of example programming problems.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License. Others are free to copy, reuse, etc. this work under the license terms at <http://creativecommons.org/licenses/by-sa/4.0/>

# Contents

<b>1</b>	<b>What is This Course About?</b>	<b>1</b>
1.1	Introduction . . . . .	1
1.2	What This Course Isn't . . . . .	2
1.3	What Are These Note About? . . . . .	4
1.4	CSci 1001 and Liberal Education . . . . .	5
1.4.1	Why Liberal Education? . . . . .	5
1.4.2	Mathematical Thinking . . . . .	5
1.4.3	Technology and Society . . . . .	7
1.4.4	How These Requirements Will Appear in the Coursework . . . . .	9
1.5	Course Structure . . . . .	10
1.6	Tips For Doing Well . . . . .	11
1.7	Additional Questions . . . . .	12
<b>2</b>	<b>Algorithms</b>	<b>13</b>
2.1	Introduction . . . . .	13
2.1.1	Introductory Problem . . . . .	13
2.1.2	Introductory Comments . . . . .	14
2.1.3	Motivation . . . . .	14
2.1.4	Skills . . . . .	15
2.1.5	Algorithms and the Liberal Education Requirements . . . . .	15
2.2	Specifying Algorithms . . . . .	16
2.2.1	An Example . . . . .	16
2.2.2	Algorithm Characteristics . . . . .	17
2.2.3	Pseudocode Characteristics . . . . .	18
2.3	Pseudocode . . . . .	19
2.3.1	Sequence, Selection, Repetition . . . . .	19
2.3.2	Indentation . . . . .	19
2.3.3	Other Basic Pseudocode Commands . . . . .	20
2.4	Additional Comments on Pseudocode . . . . .	21
2.4.1	Different Ways of Expressing an Operation . . . . .	21
2.4.2	What is a Variable? . . . . .	21
2.4.3	Lists, etc. . . . .	22
2.4.4	Variable Names . . . . .	23

2.4.5	Pseudocode Summary . . . . .	24
2.5	Some Practice . . . . .	24
2.5.1	Writing Pseudocode Fragments . . . . .	24
2.5.2	Reading an Algorithm . . . . .	25
2.5.3	Common Mistakes . . . . .	25
2.6	Two Examples For Class . . . . .	26
2.7	Writing Algorithms: How to Begin? . . . . .	27
2.8	Questions to Think About For Class . . . . .	29
2.9	Additional Practice . . . . .	29
2.10	Problem Solutions . . . . .	30
2.10.1	Introductory Problem . . . . .	30
2.10.2	Additional Problems . . . . .	30
<b>3</b>	<b>Data Representation</b> . . . . .	<b>35</b>
3.1	Introductory Problem . . . . .	35
3.2	Overview . . . . .	35
3.2.1	Goals . . . . .	36
3.2.2	Data Representation and Mathematics . . . . .	37
3.2.3	Data Representation and Society and Technology . . . . .	37
3.3	Representation Basics . . . . .	38
3.3.1	Introduction . . . . .	38
3.3.2	Why Binary? . . . . .	38
3.3.3	Review of the Decimal Number System . . . . .	39
3.3.4	Unsigned Binary . . . . .	39
3.3.5	Decimal to Binary Conversion . . . . .	40
3.3.6	Addition of Binary Numbers . . . . .	42
3.3.7	Subtraction of Binary Numbers . . . . .	43
3.4	Other Representations Related to Binary . . . . .	43
3.4.1	Hexadecimal . . . . .	44
3.4.2	Sign/Magnitude Notation . . . . .	46
3.5	Two's Complement Representation . . . . .	48
3.5.1	Two's Complement Representation Basics . . . . .	48
3.5.2	Addition and Subtraction . . . . .	49
3.6	Range . . . . .	51
3.7	Floating Point Numbers . . . . .	53
3.8	Text . . . . .	54
3.9	Bytes, Kilobytes, Megabytes, and More . . . . .	56
3.10	Image Files, Audio Files, and Video Files . . . . .	57
3.10.1	Images . . . . .	57
3.10.2	Video . . . . .	59
3.10.3	Audio . . . . .	59
3.11	Additional Problems . . . . .	60

3.12	Problem Solutions . . . . .	61
3.13	Additional Resources . . . . .	63
<b>4</b>	<b>Logic</b>	<b>65</b>
4.1	Introduction . . . . .	65
4.1.1	Introductory Puzzles . . . . .	65
4.1.2	Overview . . . . .	67
4.1.3	Why is Logic Important? . . . . .	67
4.1.4	Relation to the Mathematics Liberal Education Requirement . . . . .	68
4.1.5	Relation to the Society and Technology Requirement . . . . .	68
4.1.6	Goals . . . . .	69
4.2	Logical Operators and Low-Level Logic . . . . .	69
4.2.1	Introduction to Logical Operators . . . . .	69
4.2.2	Logical AND . . . . .	70
4.2.3	Logical OR and XOR . . . . .	71
4.2.4	Logical NOT, Equivalence, and Implication . . . . .	72
4.3	Compound Logical Statements . . . . .	74
4.4	More Bitwise Operations . . . . .	77
4.5	Information Reliability . . . . .	78
4.6	Things to Think About . . . . .	79
4.7	Additional Problems . . . . .	79
4.8	Problem Solutions . . . . .	81
4.9	Additional Resources . . . . .	86
<b>5</b>	<b>Machine Organization</b>	<b>89</b>
5.1	Introduction . . . . .	89
5.1.1	Introductory Problem . . . . .	89
5.1.2	Overview . . . . .	89
5.1.3	Motivation . . . . .	90
5.1.4	How This Topic Relates to the Mathematics Requirement . . . . .	91
5.1.5	How This Topic Relates to the Society and Technology Theme . . . . .	91
5.1.6	Goals . . . . .	91
5.2	An Analogy . . . . .	92
5.3	Chapter Structure . . . . .	93
5.4	Machine Organization Terms . . . . .	95
5.5	Memory Addresses . . . . .	97
5.6	Machine Instructions . . . . .	101
5.7	Some Quantities . . . . .	104
5.8	System Software . . . . .	105
5.9	Some Additional Notes . . . . .	106
5.10	Further Questions . . . . .	106
5.11	Additional Problems . . . . .	107

5.12	Problem Solutions . . . . .	109
5.13	Additional Resources . . . . .	113
<b>6</b>	<b>Moore's Law</b>	<b>115</b>
6.1	Introduction . . . . .	115
6.1.1	Introductory Problems . . . . .	115
6.1.2	Introductory Explanation . . . . .	115
6.1.3	Why is Moore's Law Important? . . . . .	117
6.1.4	Topic Goals . . . . .	118
6.1.5	Moore's Law and the Liberal Education Requirements . . . . .	118
6.2	More About Exponential Growth . . . . .	119
6.3	How Long Will Moore's Law Continue to Hold? . . . . .	120
6.4	Questions . . . . .	121
6.5	Problem Solutions . . . . .	121
6.6	Additional Resources . . . . .	121
<b>7</b>	<b>Computer Security</b>	<b>123</b>
7.1	Introduction . . . . .	123
7.1.1	Introductory Problem . . . . .	123
7.1.2	Overview . . . . .	123
7.1.3	Topic Goals . . . . .	124
7.1.4	How These Topics Relate to Mathematics . . . . .	124
7.1.5	How These Topics Relate to Society and Technology . . . . .	124
7.2	Some Computer Security Principles . . . . .	124
7.3	Introduction to Cryptography . . . . .	126
7.4	Public Key Encryption . . . . .	128
7.5	Additional Problems . . . . .	131
7.6	Problem Solutions . . . . .	132
7.7	Questions . . . . .	134
7.8	Additional Resources . . . . .	134
<b>8</b>	<b>Computer Science, Numbers, and Counting</b>	<b>137</b>
8.1	Introduction . . . . .	137
8.1.1	Introductory Problem . . . . .	137
8.1.2	Overview . . . . .	138
8.1.3	What Are Counting Problems and Why Are They Important? . . .	139
8.1.4	The Liberal Education Requirements . . . . .	139
8.1.5	Goals . . . . .	140
8.2	Solving Counting Problems . . . . .	141
8.2.1	Some General Tips . . . . .	141
8.2.2	Example of Solving a Problem . . . . .	142
8.2.3	Important Quantities . . . . .	144
8.2.4	A Useful Formula . . . . .	145

8.3	Example Problems . . . . .	146
8.4	Problem Solutions . . . . .	147
8.5	Additional Questions to Think About . . . . .	148
8.5.1	Some Further, Related Problems . . . . .	148
8.6	Additional Resources . . . . .	150
<b>9</b>	<b>Algorithmic Complexity</b>	<b>151</b>
9.1	Introduction . . . . .	151
9.1.1	Introductory Problem . . . . .	151
9.1.2	Overview and Motivation . . . . .	152
9.1.3	Further Explanation of Algorithmic Complexity . . . . .	152
9.1.4	Topic Goals . . . . .	154
9.1.5	Algorithmic Complexity and Mathematics . . . . .	154
9.1.6	Algorithmic Complexity and Technology and Society . . . . .	155
9.2	How To Measure Algorithmic Complexity? . . . . .	155
9.2.1	Big-O and Big- $\Theta$ . . . . .	155
9.2.2	Simplifying Big-O and Big- $\Theta$ notation . . . . .	157
9.3	Logarithms . . . . .	158
9.4	Calculating Complexities . . . . .	160
9.5	Additional Example Problems . . . . .	162
9.6	The Traveling Salesman Problem . . . . .	165
9.7	What If a Problem Takes Too Long To Solve? . . . . .	166
9.8	Problem Solutions . . . . .	167
9.8.1	Introductory Problem Solutions . . . . .	167
9.8.2	Example Problem Solutions . . . . .	168
9.9	Some Further Questions . . . . .	169
<b>10</b>	<b>What is Computer Science?</b>	<b>171</b>
10.1	Introduction . . . . .	171
10.1.1	Introductory Problem . . . . .	171
10.1.2	Overview . . . . .	171
10.1.3	Motivation . . . . .	172
10.1.4	Goals . . . . .	172
10.1.5	Connection with Mathematics, and with Technology and Society . . . . .	172
10.2	What is Computer Science? . . . . .	172
10.3	Closely Related Fields . . . . .	174
<b>11</b>	<b>Software Development</b>	<b>179</b>
11.1	Introduction . . . . .	179
11.1.1	Introductory Problem . . . . .	179
11.1.2	Overview . . . . .	179
11.1.3	Motivation . . . . .	179
11.1.4	Goals . . . . .	180

11.1.5	Connection with Mathematics, and with Technology and Society . . .	181
11.2	Programming “In the Large” Versus “In the Small” . . . . .	181
11.3	Parts of the Software Development Process . . . . .	182
11.4	Software Challenges and Risks . . . . .	185
11.5	Some Example Problems . . . . .	185
11.6	Problem Solutions . . . . .	186
<b>12</b>	<b>Python Reference</b>	<b>189</b>
12.1	Introduction . . . . .	189
12.2	Getting Started with The Python Interpreter . . . . .	190
12.3	Basics: Data Types and Operations . . . . .	190
12.3.1	Numbers . . . . .	190
12.3.2	Text . . . . .	191
12.3.3	Booleans . . . . .	192
12.4	Variables . . . . .	192
12.4.1	Variable Names . . . . .	192
12.4.2	Assignment and Use . . . . .	192
12.5	Operators . . . . .	193
12.5.1	Arithmetic Operations . . . . .	193
12.5.2	Relational Operators . . . . .	195
12.5.3	Logical Operators . . . . .	195
12.6	Lists and Indexing . . . . .	196
12.7	Input/Output . . . . .	197
12.8	Control Structures . . . . .	198
12.9	Functions . . . . .	201
12.10	Libraries . . . . .	205
12.11	Comments and Line Continuation . . . . .	205
12.12	More About Lists . . . . .	206
12.13	Random Numbers . . . . .	207
12.14	Examples . . . . .	208
12.15	Problem Solutions . . . . .	209
12.16	More About Using the Python Interpreter . . . . .	209
12.17	Additional Resources . . . . .	210
<b>13</b>	<b>Example Python Problems</b>	<b>211</b>
13.1	Introduction . . . . .	211
13.2	Problems . . . . .	211
13.3	Solutions . . . . .	223

# Chapter 1

## What is This Course About?

### 1.1 Introduction

Computer science. What exactly is computer science? Why — beyond the obvious reasons — is it important? What do computer scientists do? What types of problems do they work on? What approaches do they use to solve those problems? How, in general, do computer scientists think?

**Question 1.** *What do you think of when you hear “computer science?” Write a paragraph or list, or draw an image or diagram of what comes to mind.*

**Question 2.** *What are the parts of computer science that are most interesting or important to you currently? Why?*

When you hear the term “computer science” perhaps you think of a specific computer. Or someone you know who works with computers. Or a particular computer use, say online games or social networks. There are many, many different aspects of computing and computer science.

Furthermore, there are a number of reasons why it is useful and important to know something about computer science. Computers are affecting our lives in many different ways. For most of us, computers are playing or will play a significant role in the work we do, in our recreational pursuits, in how we communicate with others, in our education, in our health care, etc. Think about the ways you encounter computers and computing, either directly or indirectly, in your daily life.

What, more specifically, will this course cover? The foremost purpose of this course is to give you a greater understanding of the fundamentals of computer science: What is computer science, anyway? Is the the same as computer programming? What is a computer? For example, most people would agree that a “laptop computer” is a computer, as is a “tablet computer;” but what about a smartphone? And how do computers work? For example, we can store not only numbers and text in computers, but also images, video files, and audio files; how do computers handle such disparate data? And what are some interesting and important subareas of computer science? For example, what is important to know about subareas such as computer graphics, networking, or databases? And why

is any of this important? Isn't it sufficient for most people just to use computers, rather than have a deeper understanding of computers and computer science?

These are all fundamental questions about computing, and in this course we'll look at them and other questions. In summary, one purpose of this course is to provide an overview of computer science that not only exposes you to computer science fundamentals — such as how a computer works on a rudimentary level — but also explores why these fundamentals are important.

There are two parts of this overview that deserve further explanation. This course fulfills the University of Minnesota liberal education mathematical thinking core requirement and the technology and society theme requirement. So while the main theme of this course is an overview of computer science, two essential subthemes are how mathematics is used in computer science, and how computer science affects, and is affected by, society.

Both subthemes fit well in an overview of computer science course. Computer science relies heavily on mathematics (in fact, some colleges have computer science and mathematics programs in a joint department). Certain uses of mathematics in computer science are obvious — for example, in computational tools such as spreadsheets — but there are also many less obvious ways that mathematics is essential to computer science. For example at the lowest level in a computer, data (whether that data is numeric, text, audio, video, etc.) is all represented in binary, i.e., as strings of 0's and 1's. This means that to understand something very basic about computers you need to understand binary numbers and operations.

Computers are also affecting society in many ways, from the use of computer-generated imagery in films, to large government or commercial databases, to the multiple societal effects of the Internet. And society is affecting computers, for example through user behavior and through different types of regulation.

While mathematics and technology and society might seem too different to be included comfortably in the same course, there are actually many computer science topics that are useful to explore from both perspectives — in a sense, these different viewpoints are “two sides of the same coin.” For example, one topic in the course is computer security. Mathematics plays a role in security, for example in encryption. And computer security also has many societal aspects, for example national security, infrastructure security, and individual security. Most of the topics in this course similarly have both mathematical underpinnings and societal aspects, and exploring these topics from both perspectives will result in a richer understanding.

## 1.2 What This Course Isn't

There are a number of different types of introductory computer science courses. So, in addition to explaining what this course is, it is also useful to state what it is not.

*This is not a programming course.* Programming is a central activity in computer science, but it is not the whole of computer science. Because programming is important, we'll spend some time on it. However, because computer science is much more than

programming, and because this is an overview course, that time will be only a small part of the course — probably a few weeks.

If you wish to take a programming course, the University of Minnesota, like most other colleges and universities, offers a number of different introductory programming courses.<sup>1</sup>

*This is not a computer applications course.* Many colleges and universities have courses that cover basic computer applications. For example, a popular choice is teaching how to use a word processor, a spreadsheet, a database management program, and presentation software. These and other applications are important parts of computer science, and so in this course you will get a chance to work with some applications that might be new to you. However — like programming — using applications is only part of learning about computer science, and so application use will be only a small part of this course.

*This is not a “computer literacy” or “computer fluency” course.* There are a variety of definitions of computer literacy or computer fluency. For example the Wikipedia definition, derived from a report from the U.S. Congress of Technology Assessment, is “the knowledge and ability to use computers and related technology efficiently, with a range of skills covering levels from elementary use to programming and advanced problem solving.”<sup>2</sup> Parts of this course will involve using computers to gain a variety of skills. For example, in the labs and homeworks in this course you will do a variety of computer-related tasks such as performing web searches, constructing web pages, doing elementary computer programming, and working with databases. However, this is just one part, rather than the totality, of the course. So this course shares some characteristics of a computer literacy course, but overall it has a wider focus than that type of course.

*This is not a “great ideas in computer science” course.* One current trend in computer science introductory courses is to study computer science through its important, fundamental ideas.<sup>3</sup> And this course does cover some key ideas. For example, an early topic we’ll study is how all data in computers, whether that data be numeric, text, video, etc. is represented within the computer as 0’s and 1’s. In general, the topics in the course are fundamental to computer science. However, this course also differs from a great ideas course. It is not focused solely on ideas, but explores broadly a number of computer-related issues, subtopics, and computer skills. Moreover, to fulfill the University of Minnesota liberal education requirements this course focuses more on mathematical thinking, and on technology and society, than a typical great ideas course would.

In addition to programming, applications, computer fluency, and great ideas, there are a number of other types of introductory computer science courses. Some are courses

---

<sup>1</sup>Specifically, CSci 1103 is a Java programming course for non-majors, CSci 1113 is a C++ programming course for science and engineering majors, and CSci 1133 is an introductory Python programming and computer concepts course for computer science majors.

<sup>2</sup>See [http://en.wikipedia.org/wiki/Computer\\_literacy](http://en.wikipedia.org/wiki/Computer_literacy). Accessed May 20, 2105.

<sup>3</sup>For example, see <http://denninginstitute.com/pjd/GP/GP-site/welcome.html> (accessed May 20, 2015), Peter Denning’s “Great Principles of Computer Science” website. This site organizes principles into seven categories: computation, communication, coordination, recollection, automation, evaluation, and design. There are a number of good ideas, insights, and frameworks in this and related approaches, and in fact many of the key ideas in this course will relate in some way to Denning’s principles.

that survey a variety of computer science topics. Others focus on professional software development practices. Still others look at computing through a particular “lens” such as networks or computational biology. And so on. This course has some common characteristics with these other courses, but also has significant differences. In particular, the biggest difference is this course blends an overview of computer science with a strong emphasis on mathematics, and on society and technology; this is a balance of emphases that has a number of advantages, but is not usually seen in introductory computer science courses.

### 1.3 What Are These Note About?

There is no textbook for this course. The reason for this is that although there are a number of excellent “introduction to computer science” textbooks, none is a good fit for this course. Instead, these notes are the “textbook.”

Specifically, in order to fulfill the University of Minnesota liberal education requirements, both mathematical thinking and technology and society need to be significant parts of this course. Many textbooks present an introduction to computer science through programming, or through how computers work, or through some other aspect of computing. However, there is not a suitable text that combines an overview of computer science with both sufficient mathematical and sufficient society and technology emphases.

But these notes are not a textbook in the traditional sense. For example, they are neither as long nor as detailed as a textbook. There are a few reasons for this.

One is this course has a number of different parts, and these notes are kept short so reading them doesn’t take so much time as to interfere with other course activities. Another is that there are a number of online resources such as tutorials, reference guides, and instructional videos freely available on the Internet, and that you can use as supplemental resources. (In an overview of computing course it is particularly appropriate to make use of educational resources that others have been good enough to post. Being able to learn from different resources that perhaps were created for slightly different audiences, and that might use slightly different notation, etc. is a good skill to have.) A third, related reason is that there will also be some additional required readings. These will usually be short, online readings that we will use to explore topics in more depth.

Another key distinction between these notes and a traditional textbook is that these notes often focus on fundamental or background material — material that you can often learn most efficiently from reading. Using this background material to solve various problems, or to explore technology and society issues is more difficult, and so many of the course learning activities will be done during “lecture” time or during lab, and will build on rather than repeat this fundamental material.

## 1.4 CSci 1001 and Liberal Education

CSci 1001 fulfills two University of Minnesota liberal education requirements: the mathematical thinking core requirement, and the technology and society theme requirement. This section explains how the course satisfies the criteria for these requirements.

### 1.4.1 Why Liberal Education?

At first glance, it might seem odd that a course entitled “Overview of Computer Science” fulfills liberal education requirements. What does computer science have to do with liberal education?

However, a course such as CSci 1001 is a good fit for certain liberal education requirements. Understanding computers well involves exploring them from a variety of different viewpoints. This includes understanding not only how computers work — including, for example, the mathematical underpinnings of computer science — but also how they are affecting, and are affected by society. In summary, to have a good understanding of computers and computer science it is important to explore them from a variety of perspectives, including the perspectives embodied in some of the liberal education requirements.

### 1.4.2 Mathematical Thinking

**Question 3.** *What do you think of when you hear the word “mathematics?” Write a paragraph or list, or draw an image or diagram of what comes to mind.*

**Question 4.** *Based on your experience with computers, write a list of some places where mathematics is used in computing.*

What do computers and mathematics have in common? Why is it appropriate for an overview of computer science course to satisfy the liberal education mathematical thinking requirement?

To fulfill the mathematical thinking requirement, a course must fulfill the following criteria<sup>4</sup>

- The course exhibits the dual nature of mathematics both as a body of knowledge and as a powerful tool for applications.
- Students manipulate mathematical or logical symbols.
- The prerequisite math requirements and mathematics used must be at least at levels that meet the standards for regular entry to the University.

The rest of this subsection explains how these criteria relate to the material and themes in this course.

---

<sup>4</sup>From [http://onestop.umn.edu/faculty/lib\\_eds/guidelines/mathematical\\_thinking.html](http://onestop.umn.edu/faculty/lib_eds/guidelines/mathematical_thinking.html), accessed May 20, 2015.

*The course exhibits the dual nature of mathematics both as a body of knowledge and as a powerful tool for applications.*

Much of the use of mathematics in this course is applying mathematical ideas and operations to solve computer science problems. There are a number of important mathematical underpinnings of computer science, and so understanding computer science involves being able to solve mathematical problems involving these underpinnings. At the same time, the different uses of mathematics in this course exemplify characteristics of mathematics as a whole, and of the close tie between the fields of mathematics and computer science. For instance the mathematics in the course illustrates the following:

1. The reliance of many key ideas in computer science, such as data representation, on mathematics.
2. The use of special mathematics- or logic-related notation and terminology in many parts of computer science.
3. The ability to represent and work with many different types of data in the computer, and the related ability to represent and work with quantities in different representations using a variety of operations.
4. The need for rigor in solving problems, analyzing situations, or specifying computational processes.
5. The use of numbers and arithmetic in solving computational problems. However, rather than being simple arithmetic problems, these problems often have some special characteristics such as involving repeated operations, or involving extremely large or extremely small numbers.
6. The existence of a variety of different algorithms for solving such diverse problems as pattern matching, counting specified values in a table of data, or finding the shortest path between two nodes in a graph.

*Students manipulate mathematical or logical symbols.*

Solving many of the problems in this course will involve doing some mathematics, and therefore manipulating mathematical or logical symbols. Here are a few examples:

1. In exploring low-level logical operations you'll need to manipulate binary representation and logical operators.
2. In studying the growth rate of algorithms you'll need to work with the "big- $O$ " and "big- $\Theta$ " notations commonly used by computer scientists.
3. In specifying computational processes you'll need to use "pseudocode" or a programming language. These share many notational characteristics with mathematical or logical symbols, especially when the computational processing involves a large number of numeric computations.

*The prerequisite math requirements and mathematics used must be at least at levels that meet the standards for regular entry to the University.*

The level of mathematics in this course is introductory-level college mathematics. As such, the mathematics is not advanced, and there is no mathematical prerequisite for this course beyond the requirements needed for admission to the University. At the same time, the mathematics in this course goes beyond high school mathematics even though many of the types of mathematics used in this course appear in some high school mathematics courses.

As an example, one appearance of mathematics in this course is binary (or base 2) representation. This is a topic that often appears in high school mathematics courses, and the basics of binary representation are not complicated. In this course we review such basics as how to convert numbers between decimal (base 10) and binary representation, and how to do simple operations such as adding two binary numbers. However, we also use binary representation in additional ways that underpin the workings of computers. Here are a few examples:

1. We'll look at a few different ways to represent numbers in binary representation. For example, computers usually do not use the straightforward binary representation when representing integers, but rather use "two's complement" form. So part of this course is learning not only about the "usual" binary representation, but also about these alternatives.
2. We'll look at various issues with binary representation, such as the number of "bits" used, that are important in determining the range and precision of numbers used by computers.
3. In addition to representing numbers, we will also look at how computers use binary representation to represent and operate on other types of data such as text, colors, and images.
4. In addition to basic operations such as binary addition, we will also look at other operations on binary representations. For example, logical operations are important in masking colors in image processing, and in implementing arithmetic operations in low-level computer hardware.

In summary, even though many mathematical topics in this course appear in high school mathematics, they go beyond the usual high school treatment of those topics in breadth or depth.

### 1.4.3 Technology and Society

**Question 5.** *What do you think of when you hear "technology and society?" Write a paragraph or list, or draw an image or diagram of what comes to mind.*

**Question 6.** *Based on your experience with computing, write a list of examples of how computing is affected, and being affected by, society.*

To fulfill the technology and society requirement, a course must fulfill the following criteria<sup>5</sup>

- The course examines one or more technologies that have had some measurable impact on contemporary society.
- The course builds student understanding of the science and engineering behind the technology addressed.
- Students discuss the role that society has played in fostering the development of technology as well as the response to the adoption and use of technology.
- Students consider the impact of technology from multiple perspectives that include developers, users/consumers, as well as others in society affected by the technology.
- Students develop skills in evaluating conflicting views on existing or emerging technology.
- Students engage in a process of critical evaluation that provides a framework with which to evaluate new technology in the future.

The rest of this subsection explains how these criteria relate to the material and themes in this course.

*The course examines one or more technologies that have had some measurable impact on contemporary society.*

The topic of this course is computers and computing. Computers have affected society in numerous and diverse ways, some of which we'll explore in this course. And current and future computer applications will affect society in even more ways.

*The course builds student understanding of the science and engineering behind the technology addressed.*

Through this course you should get an understanding of how computers work. This includes understanding the basics of computer hardware and computer software.

More broadly, however, computer science relies on results from other areas of science, engineering, and related fields. The most prominent example of this we will see in this course is various ways that mathematics is essential in computer science.

*Students discuss the role that society has played in fostering the development of technology as well as the response to the adoption and use of technology.*

Technology affects society. However, it is not a one-way street. Society also affects technology. For example, society fosters technology by means such as government support

---

<sup>5</sup>From [http://onestop.umn.edu/faculty/lib\\_edsguidelines/technology\\_and\\_society.html](http://onestop.umn.edu/faculty/lib_edsguidelines/technology_and_society.html), accessed May 20, 2015.

for research. As another example, individuals, businesses, and other organizations adopt and use technology in ways often not foreseen by the technology's creators.

In this course we'll look at a variety of instances of how society affects technology. These include government funding for the early Internet, Internet regulation, how business considerations affect computing products, and societal aspects of computer security.

*Students consider the impact of technology from multiple perspectives that include developers, users/consumers, as well as others in society affected by the technology.*

In many topics in computers and society there are multiple stakeholders. These can include individual users, developers, companies (producers, consumers, and intermediaries), government bodies, professional organizations, and other types of organizations. These different stakeholders often have different views and different goals.

In this course we will often look at technology and society issues from numerous perspectives. Sometimes we will focus on a specific perspective or the role of a specific stakeholder. However, other times we will explore issues more broadly: Who are the stakeholders? What is their role in this issue? What are their goals? etc.

*Students develop skills in evaluating conflicting views on existing or emerging technology.* One often hears conflicting views on computer and society issues. Computers are beneficial for society. Computers are harmful to society. The Internet is making it easier for people to communicate and is bringing people together. The Internet is making people more isolated. Computers and automation are robbing people of jobs. Computers and automation create jobs.

In this course we'll often explore issues that are contentious and/or complicated. How do we avoid a superficial, one-sided understanding of such issues? How do we resolve conflicting claims about such issues?

*Students engage in a process of critical evaluation that provides a framework with which to evaluate new technology in the future.*

Computing technology not only has had massive effects on society, but it is continuing to affect society. Not a day goes by without some technological advance involving computing. In many ways the "computer revolution" is just beginning.

One goal of this course is that you'll learn enough about computing in general, about trends in computing, and about computing and society that you'll be able to evaluate new technology. Note "evaluate" here might mean different things in different contexts. For instance, it might mean give an informed projection about whether a new computer product will be successful or not. Or it might mean predict future computer advances in a certain area. Or it might mean analyze whether a new computer application is more likely to be beneficial than harmful.

#### **1.4.4 How These Requirements Will Appear in the Coursework**

Both the mathematical thinking and technology and society requirements will appear prominently in the coursework you do. In particular, many of the individual homework

assignments will involve mathematics in some way, shape, or form. Similarly, many lab problems will also often involve mathematics. And some in-class activities will be practice for these labs and homework problems.

A few of the homework and lab problems will involve the technology and society theme. Moreover, most weeks there will be short writing and/or an in-class discussion of the technology and society aspect of the course topics.

Finally, both mathematical thinking and technology and society problems will be on the exams, with questions often similar to those on the homework or from the in-class activities or discussions.

## 1.5 Course Structure

The course has a number of components:

- *Class lectures* will explore important topics from computer science. This includes both technical aspects and computers and society aspects.
- *Technical in-class exercises* provide practice on technical aspects of the current topic. Problems will usually be mathematical to fulfill the math liberal education core criteria; however, occasional society and technology questions will also be included.
- *Discussions/exercises on society and technology* provide a chance for interactive discussion and debate of current computer science-related social issues.
- *Weekly laboratory exercises* allow hands-on exploration of course content. These lab sessions occur in a classroom laboratory where exercises can be completed by pairs of students working on computers.
- *Reading assignments* are designed to prepare you for homework, labs, exams, and discussions.
- *Written problem assignments* help you explore computer science concepts in depth. Unless otherwise stated, these assignments must be completed *individually*, and will be due about every other week.
- *Other occasional in-class or between-class assignments*, for example short writing assignments, serve a variety of functions and will be explained further in class.
- *Exams* give you a chance to demonstrate your knowledge of the course material. There will be one or two midterms exams and a final exam. See the course syllabus and/or web page for more information.

Note that the in-class exercises, labs, etc. are all important parts of the course, and will contribute to your course grade. It is therefore important that you attend class (including

the lab). It is also important that you do any assigned preliminary work, including any reading or writing, prior to lecture and lab.

Additional information on these components, as well as important administrative material such as exam and assignment rules, will be posted on the course web page.

## 1.6 Tips For Doing Well

Here are some tips for doing well in CSci 1001. Although most of these are straightforward, they are particularly relevant to a course such as this one.

- *Show up.* A large part of doing well in this course is showing up. Don't miss class unless you have a valid excuse (such as illness). And if you do miss class then check with others to see what you have missed.

The labs, in-class exercises, and discussions are all important parts of the course. Sometimes they are important learning activities in and of themselves; other times their purpose is practice to help on the homework problems and exams. To emphasize their importance, a portion of the class grade is devoted to the labs, and to in-class activities such as in-class exercises or discussions.

- *Start the homework early.* Most homework will be posted a couple weeks before it is due. Usually when it is posted you will have seen enough material to start at least some of the problems. Starting early will give you enough time to think about the more difficult questions, and to ask questions during office hours.
- *Come to office hours if you have questions.* If you have any questions on the homework, or are having trouble with it, please come to office hours.
- *Do the reading.* We will usually assume you've done the assigned reading, and done it before class. Sometimes we'll use class time to go over some particularly important and/or challenging parts of the reading. Other times the class lecture will use the reading as a starting point, but not re-explain it in detail.
- *Get to know others in the class.* Many people learn better if they discuss class material with others. Get to know people in the class, form study groups, etc. Some of the assignments in the class — notably the labs — are designed as group assignments. Others such as the homework and exams are individual work; however, even on these you are welcome to do preliminary studying in groups, but your answers on the assignment and exams must be yours alone. (See the further explanation on the class web page for more details.)
- *Use the web resources.* Throughout the semester we will post additional resources to the course web page. Moreover, a number of other online resources are mentioned in these notes. See which of these are most useful to you, and use them accordingly.

- *Realize that some material in this course might be easy, but some might be hard.* Students in this class come from a variety of backgrounds. Often, students will find some parts of the course easy, but then find other parts require significantly more time and effort.
- *Persist:* Many students will find at least part of this course to be challenging. If you have not seen, for example, topics such as algorithms before, they will seem foreign and will take time to master. Do the reading, ask questions as needed, and practice doing problems.
- *Try to apply this material to your major or to other interests:* Much of the information or skills in this course are applicable to a wide variety of areas. Think about how what you are learning in this course might be applicable to other courses you are taking, or to other areas of your interest.

## 1.7 Additional Questions

Here are additional introductory questions, some of which will likely be used for between-class or in-class exercises or discussion.

**Question 7.** *How do you use computers? List the most important ways.*

**Question 8.** *Write down a list of movies in which computing plays a major role. For each movie, indicate whether computing is portrayed as beneficial, harmful, beneficial in some ways but harmful in others, or neutral.*

**Question 9.** *Do you think computers, on the whole, have more positive effects than negative ones, more negative ones than positive, or about equal positive and negative effects? Why?*

**Question 10.** *List some ways computers are beneficial to society. Then list some ways they are harmful.*

**Question 11.** *Suppose you were to write a novel, play, screenplay, etc. about some aspect of computers and society. Describe what the theme or themes of your work would be.*

**Question 12.** *What does technology mean? What are some important ways you use technology in your daily life?*

**Question 13.** *Suppose you had to write a short essay or short story entitled “Computers and Me.” What would be some key points or themes in that work?*

**Question 14.** *Suppose you had to write a short essay or short story entitled “Technology and Me.” What would be some key points or themes in that work?*

# Chapter 2

## Algorithms

Precisely, step by step.

### 2.1 Introduction

#### 2.1.1 Introductory Problem

Consider the following problem:<sup>1</sup>

Mobile robots must navigate through their environment without bumping into obstacles. Consider the following obstacle avoidance problem. Suppose you have a very simple rectangular maze. There's a designated start square, a designated finish square, a single path from start to finish, and no dead ends. Suppose you also have a robot that can do the following:

- **moveForward**: move forward one square.
- **turnLeft**: turn ninety degrees to its left.
- **turnRight**: turn ninety degrees to its right.
- **startInMaze**: this places the robot at the start square, and orients it so its first valid move is straight ahead.
- **checkForWall**: check if there is a wall immediately ahead, and return true if there is and false if there isn't.
- **checkForMazeEnd**: this checks if the robot is at the end square, and returns true if it is and false if it isn't.

---

<sup>1</sup>These notes will often start a chapter with a problem from a previous offering of CSci 1001. These problems will give you an introduction to the chapter topic, as well as an example of some types of problems that might appear in the homework or exams. A solution to the introductory problem will usually, but not always, appear at the end of the chapter.

Using a correct combinations of these, along with other basic operations such as `get` (for input), `set` (to assign a name to a value used in the program), or `print` (for output), devise an algorithm that gets a maze as input, places the robot on the start square, and navigates the robot through the maze. Once the robot reaches the end, the algorithm should print out a message stating it is at the end, and another stating how many moves it made navigating the maze.

### 2.1.2 Introductory Comments

Some universities have a class on “Great Ideas in Computer Science.” As mentioned in Chapter 1, our class differs from these great ideas classes in significant ways. However, it also has similarities. In particular, the topics in this class focus on key ideas that make computer technology and practice possible.

The first key idea we’ll explore is that of an *algorithm*. Roughly speaking, an algorithm is a set of precise instructions for solving a problem. (We’ll look at a more specific definition in class.) This concept is essential because accomplishing any task with a computer requires clearly and unambiguously specifying the steps a computer should perform to complete the task. Because this is so central to what computers do, we will use algorithms again and again in this class.

Why, more specifically, are algorithms important? How do they appear in this class? How do they relate to the course’s liberal education requirements? What should you be able to do with algorithms? How do people represent algorithms? What is the connection between algorithms and computer programs? This chapter addresses these and related questions.

### 2.1.3 Motivation

Algorithms might seem like an odd starting topic for this class. Are algorithms really that important?

They are. In fact, some computer scientists see algorithms as *the* central concept in computer science. As mentioned above, before solving a problem with a computer there must be a precise specification of the steps the computer must perform. This precise specification is an algorithm. Computer programs are implementations of algorithms, so algorithms underlie programming. To understand how efficient a computer solution to a problem is, computer scientists analyze algorithms. To create a more efficient solution computer scientists try to improve existing algorithms or discover alternative algorithms. (Or they might prove that a more efficient algorithm cannot exist.) There are some algorithms that are important generally, such as algorithms for searching for an item in a list, or sorting all items in a list.<sup>2</sup> And there are important algorithms for subareas of

---

<sup>2</sup>In fact, there are a number of different searching and sorting algorithms; for example, think about how many different ways you can put a shuffled deck of cards in order.

computer science, for instance algorithms for coloring and rendering shapes in computer graphics, and algorithms for merging two different database tables.

Algorithms will occur throughout much of this class; for example, we will see algorithms again in the chapters on algorithmic complexity and computer programming.

### 2.1.4 Skills

Once we complete this topic, you should be able to do the following:

1. Be able to explain what an algorithm is and isn't, why algorithms are important to computer science, and how algorithms are usually represented.
2. Given a purported algorithm, be able to determine whether it is indeed a valid algorithm; if it is not, be able to say why it is deficient.
3. Given an incorrect algorithm or partially complete algorithm, be able to identify any errors, and correct or complete the algorithm.
4. Given an algorithm, be able to trace through it and explain what it is doing.
5. Given an algorithm to solve one problem, and given a second, related problem, be able to modify the algorithm to solve the related problem.
6. Given a problem whose solution can be expressed as an algorithm, write a correct and valid algorithm to solve that problem.

### 2.1.5 Algorithms and the Liberal Education Requirements

How do algorithms embody the liberal education requirements? While algorithms are not the usual type of mathematics like algebra or calculus, they nonetheless exhibit many mathematical characteristics and require a variety of mathematical skills. For example:

- Algorithms require the specificity, clarity, and attention to detail that is a characteristic of mathematics.
- Algorithms use special keywords, notation, or conventions (such as indentation to indicate algorithm structure). This is similar to the use of special notation, etc. in mathematics.
- Algorithms describe computational processes. Specifically, an algorithm describes a procedure for doing a sequence of computations. Often individual computations are simple (for example, simple additions or comparisons rather than sophisticated mathematical functions), but the entire sequence constitutes a complicated computational process.

Algorithms are also related to the society and technology theme. The readings, discussions, and the occasional homework and/or lab questions ask you to think about such questions as

- What types of tasks can computers do and what can't they do? Or, put another way, what types of tasks can be solved by algorithms, and what types cannot?
- Is increasing automation a benefit or a concern?
- What are some current societal problems that computer practitioners are trying to solve by finding new algorithms or improving existing ones?

## 2.2 Specifying Algorithms

### 2.2.1 An Example

Suppose you write software for a construction firm. For a given construction project, the software maintains a list of on the job injuries: how many injuries occurred the first day, how many the second day, etc. You need to write a function that goes through all the days and counts the number of days that no injuries occurred. (Note different projects might have different total numbers of days. Since we need to be flexible, let  $n$  stand for the total number of days for any given project.) Here is a description of an algorithm for that task:

#### Algorithm 1

*Input:* A total number of days  $n$ , and a list  $A$ , with  $A[i]$  giving the number of on-the-job injuries on day  $i$ .

*Output:* A message stating the number of days with 0 on-the-job injuries.

```

1  Get n
2  Get A[1], ..., A[n]
3  Set i to 1
4  Set countZeros to 0
5  While i <= n
6      If A[i] equals 0, then
7          Set countZeros to countZeros + 1
8      Set i to i + 1
9  Print 'The number of zeros in the list is ', countZeros
10 Stop
```

This is a description of an algorithm for solving the problem. The algorithm is given in *pseudocode*. Pseudocode, as the name suggests, is somewhat like programming code, but not quite. Computer scientists often use pseudocode rather than a natural language (such

as English) description because natural language is usually imprecise. And they usually use pseudocode rather than an actual programming language for a number of reasons, including that pseudocode avoids many language rules programmers need to remember when writing program code. For example, some programming languages require a line of code to end with specific punctuation such as a semicolon. Remembering this (and remembering the exceptions where a semicolon is forbidden) is an extra burden we would like to avoid when focusing only on the steps the computer must do to solve the problem.

**Problem 1:** Think about some other reasons why pseudocode is often preferable to natural language and to programming code for algorithm specification.

### 2.2.2 Algorithm Characteristics

Before exploring pseudocode further, let's return to the question "what is an algorithm?" Recall that, generally speaking, an algorithm is a specific set of instructions for solving a given problem. More specifically, though, algorithms have certain characteristics:

1. *Input specified.* The algorithm must specify any input. Note in the example above the input is the number of days as well as the list containing the number of injuries for each day. Two additional notes: First, since there are many different possible types of input, it is often important to specify not only what the input is, but also its type (examples: "a string  $S$  containing alphabetic characters and digits," and "a list  $A$  of nonnegative integers"). Second, occasionally we will omit the input specification for an algorithm for the sake of brevity. However, in your course work please include the input specification unless otherwise instructed.
2. *Output specified.* The algorithm must specify any output. Usually the output is a number, string, message, list, or some combination of these. As with input, occasionally we will omit the output specification; however, you should always include it unless instructed otherwise.
3. *Correctness.* This characteristic is straightforward: an algorithm must solve the problem correctly for all possible valid input.
4. *Finite.* The algorithm finishes in a finite amount of time. Put another way, the algorithm will always solve the problem and stop.
5. *Precise.* Each step in the algorithm is precise, to the point it should need no further explanation or expansion. Moreover, each step in the algorithm is doable by a computer. Note that the algorithm above avoids instructions such as "Get the input", "Find all the 0's in the list,", "Output the message", or other instructions that are ambiguous or insufficiently precise.

Put another way, if the pseudocode for an algorithm is given to a programmer, he or she should have no questions about how the algorithm works. For that reason,

turning pseudocode into program code is often straightforward (as computer tasks go).

6. *Generality.* The algorithm isn't so specific that it solves the problem only under certain unnecessary restrictions. For example, the algorithm above solves the problem for a general number of days  $n$ . The number is not restricted to a single value — an algorithm that solved the problem only for projects that lasted, say, exactly 14 days would not be very useful.<sup>3</sup>

**Problem 2:** Consider the following set of instructions. It is a valid algorithm or not? If it is not, state which characteristic or characteristics it does not possess.

**(Purported) Algorithm 2**

*Input:* A total number of days  $n$ , and a list  $A$ , with  $A[i]$  giving the maximum temperature in degrees Fahrenheit on day  $i$ .

```

1  Get n
2  Get A[1], ..., A[n]
3  Set maxTemp to A[1]
4  Set i to 2
5  While i <= n
6      If A[i] > maxTemp
7          Set maxTemp to A[i]
8  Print 'The maximum temperature was ', maxTemp
9  Stop
```

### 2.2.3 Pseudocode Characteristics

Note some characteristics of the pseudocode description in Algorithm 1:

- *It is highly structured.* It contains a sequence of operations along with the control instructions `if` and `while`.
- *It contains a mixture of English and operations.* Specifically, it contains some English words such as `get`, `if`, and `print`, and some mathematical notation and operations such as  $n$ ,  $A[1]$ , and addition.
- *All the instruction in it are low-level.* Put another way, each step is specific — specific enough that a person should be able to perform the instructions without further explanation.

---

<sup>3</sup>Actually the algorithm is more general than stated — it counts the number of zeros in an arbitrary list of numbers regardless of whether the list holds number of on-the-job injuries on a construction project, maximum temperatures, number of diabetes-related hospital admissions, numbers of times you played *Tetris*, etc.

These characteristics describe pseudocode: it is sufficiently specific and low-level to be used in algorithm specification, but contains a simple structure and enough English constructs that it is easier to read and understand than computer code. In the next section we'll look at the details of pseudocode.

## 2.3 Pseudocode

Learning the basics of pseudocode is not difficult. Instead the challenge is using the basics to specify algorithms. This section presents pseudocode basics. In subsequent sections we'll look at using the basics.

Before beginning, we should note there are many different versions of pseudocode. All versions allow you to specify any algorithm, so all versions have much in common. However, they often differ in instruction names or other conventions. The version of pseudocode in this class is a simple one. However, be aware that different textbooks might use other versions.

### 2.3.1 Sequence, Selection, Repetition

To describe an algorithm, you need three basic control mechanisms:

- *Sequence.* Unless otherwise specified, steps are followed in sequence, one after another. So the algorithm above executes Line 1, then Line 2, then Line 3, etc.
- *Selection.* There are times when we want steps to be performed only if a certain condition or conditions hold. Line 7 in Algorithm 1 is done only if the condition in Line 6 is true. The `if` statement in Line 6 is an example of a selection statement.
- *Repetition.* Often there are parts of algorithms that are repeated many times. An example of this is the “while loop” in Lines 5–8 in Algorithm 1; this loop allows the algorithm to check each number in the list.

Pseudocode must include these three control mechanisms. Sequence is easy — simply list the steps in order. (Note line numbers are not absolutely necessary, so we will include them only when useful.) Selection uses the `if` construct and its variants such as `if-else`. Repetition uses `while` loops or `for` loops.

### 2.3.2 Indentation

Notice the indentation in Algorithm 1. This indicates the “scope” of the `while` and `if` instructions there. Specifically, the indentation of Lines 6 through 8 indicates the “body” of the while loop that starts in Line 5. The indentation lets a reader know that the instructions in Lines 6 through 8 should be performed as long as the condition in Line

5 is true. Once Line 5 is executed when that condition is no longer true, the algorithm skips to Line 9.<sup>4</sup>

Notice also the further indentation in Line 7. This is the body of the `if` statement. The indentation indicates that this statement should be executed only if the condition in Line 6 is true. If the condition is false, then the algorithm skips Line 7 and resumes with Line 8.

Line 7 is a single-statement body of the `if` instruction. Multiple-statement bodies, such as the one for the `while` loop, are also possible for `if` statements. For example, suppose each time a 0 was found you wanted the algorithm not only to update the count, but also to print a short message. Then you could include another line right after Line 7, with the same level of indentation:

```
6     If A[i] equals 0, then
7         Set countZeros to countZeros + 1
7b        Print 'No injuries on day ', i
8     Set i to i + 1
```

Lines 7 and 7b would be executed if the condition in Line 6 is true. Otherwise, neither would be executed, and the algorithm would skip to Line 8.

As Algorithm 1 indicates, you can ‘nest’ control structures. In particular, it has an `if` statement within a `while` loop. Pseudocode has only a few basic building blocks, but can describe very complicated algorithms because of the ways you can combine these building blocks. You can place loops within loops (this is useful, for example, if you are looping through a table: one loop will step through the columns, the other through the rows), `if`’s within `if`’s (useful when you have complicated selection conditions), loops within `if`’s or `if`’s within loops, etc.

### 2.3.3 Other Basic Pseudocode Commands

In addition to `if` statements and their variants such as `if-else`, and `while` and `for` statements,<sup>5</sup> pseudocode has a few other basic building blocks. Each of these appears in Algorithm 1 above.

- `get` allows the algorithm to obtain input.
- `set` allows the algorithm to assign a value to a variable.
- `print` allows the algorithm to output a message or the value of one or more variables.

---

<sup>4</sup>Note that the loop condition in Line 5 must become false at some point for the algorithm to continue past the loop and eventually stop. A loop that never stops is called an “infinite loop.”

<sup>5</sup>The `if-else` and `for` statements haven’t been discussed yet. Problem 3 below contains an example of `if-else` use. However, `for` loops won’t appear until later in the class.

- **stop** indicates the termination of the algorithm. Usually it is the last line of the algorithm, but complicated algorithms may have more than one stop. For example, it is possible to have a stop statement at the end of an **if**-body.<sup>6</sup>

In addition to these basic building blocks, pseudocode can use basic arithmetic and other operations. Algorithm 1 contains arithmetic comparisons in Lines 5 and 6, and simple additions in Lines 7 and 8. Pseudocode can also include other operations including subtraction, multiplication, division, exponentiation, and finding remainders; using functions such as square roots, sines and cosines, and logarithms; comparing alphabetic characters, or concatenating two alphabetic strings.

## 2.4 Additional Comments on Pseudocode

This section contains additional comments on a few potentially confusing aspects of pseudocode.

### 2.4.1 Different Ways of Expressing an Operation

There are a number of different ways to express the same concept using pseudocode. For example, suppose we want to set the value of the variable *s* to 42. Ways to write this include

```
s = 42
Set s = 42
Set s to 42
Set the value of s to 42
s ← 42
```

Whatever way you write operations like this should be precise, and should be used consistently; but otherwise use whichever alternative you feel most comfortable with. Moreover, when you are given an algorithm, recognize that it might use slightly different terms and notation than you've seen in class.

### 2.4.2 What is a Variable?

The language surrounding “variables” can be a little complicated. Here are two issues. The next subsection discusses an additional one.

First, variables can hold any values that can be represented within a computer. Often this is a number, e.g., **Set x to 42**. However, it could also be a list or a character string, for example **Set A to the list 1,2,3,4,5** or **Set A to the string "TTCCAGC"**, or

---

<sup>6</sup>Some versions of pseudocode omit the **stop** command with the understanding that an algorithm stops when it runs out of lines to execute.

Set `A` to the string `"string"`. Notice in all these cases there is a difference between the variable *name*, and the variable *value*. For example, in `Set x to 42` the variable name is `x`, while the value is `42`.

Second, we indicate strings by enclosing them in single or double quotes. So, e.g., the statement `Set a to 'b'` means set the value of the variable `a` to the single character `'b'`. However, the statement `Set a to b` means set the value of the variable `a` to the value of the variable `b`. Note the reason for the possible confusion here — any character or string can be a variable name, or it can be a character value assigned to a variable. It's important to keep these two different uses straight.

**Problem 3:** To explore the difference between variables names and values further, trace through the following code and figure out what it prints:

```

1 Set a to 'x'
2 Set b to 'y'
3 If a equals b then
4   Set a to 'b'
5 Else
6   Set a to b
7 Print a
8 Stop

```

The answer is at the end of this chapter.

### 2.4.3 Lists, etc.

Lists, strings, and tables have a variable name (e.g., `A` in Algorithm 1 above), but contain a number of values. So when working with them we also need to keep track of a string location, list position, etc. Consider `Set myString to "CGATG"`. Then the second item in the string is the character `'G'`. Notice that if we want to refer to that item we can't just say "the character `'G'` in the string" since there can be more than one `'G'`. So in this case there is the string name, `A`, the location in the string, `2`, and the value at that location, `'G'`.

One additional caution: there are different terms for 'location'. Some algorithms will reference the second location in the string `myString` by saying 'index 2', 'slot 2', 'subscript 2', 'location 2', 'entry 2', 'item 2', 'position 2', `myString[2]`, `myString2` and so on. The lack of a standard term and notation for this can be confusing. However if you keep the key point here in mind — that in a list or string there is a name, a location (or index, etc.), and the value in the list or string at that location — you'll find it easier to work with lists and strings.

### 2.4.4 Variable Names

In pseudocode (as in most programming languages) you can name variables anything reasonable. Here are some general guidelines.

- *Descriptive names:* Use descriptive variable names. For example, if you are counting the number of times something occurs in a list or table, then the variable name is usually something like `count`. Longer or more descriptive names are possible, including multiword names such as `countZeros` or `count_zeros` (note these use capitalization and the underscore, respectively — you should not have a space in a variable name). You don't want to get carried away with too long of a name, but using a descriptive name can help keep variables straight, and make an algorithm easier to read.

As an example, suppose you are working with a table and need to keep track of the current row and column. So you could call the related variables `row` and `column`, or `r` and `c`. Both these choices are more descriptive to someone reading the algorithm than, say, `a` and `b`, or `x` and `y`.<sup>7</sup>

- *Loop control variables:* `i` and `j` are commonly used as loop control variables, e.g., `While i < 10`.
- `n`: The variable `n` is often used to hold an input value, as in the following:

```
Get n
Print 'n squared is ', n * n
```

A second common use is to indicate the length of a list (or some other upper limit). Suppose you have a list of the number of hours per day you worked over an  $n$ -day period. The following algorithm computes the total number of hours you worked:

**Algorithm 3: Sum the numbers in a list**

*Input:* A total number of days  $n$ , and a list of numbers  $A$ .

*Output:* A message stating the sum of all the numbers in the list  $A$ .

```
Get n
Get A[1], ..., A[n]
Set i to 1
Set sum to 0
While i <= n
```

---

<sup>7</sup>As another example, rewrite the pseudocode in Section 2.4.2 by replacing the variable names  $a$  and  $b$  by  $firstChar$  and  $secondChar$ , respectively. But leave the value 'b' in Line 4 as it is. The pseudocode should be easier to understand.

```
    Set sum to sum + A[i]
    Set i to i+1
Print 'The list sum is ', sum
Stop
```

### 2.4.5 Pseudocode Summary

In pseudocode you do not have a large number of basic building blocks. You have a way of getting input, a `get` statement. You have a way of assigning values to variables, a `set` statement. You have the usual low-level operations on numbers, characters, strings, and lists. For example, you can add two numbers, compare two numbers, etc. You have an `if` statement, and variants such as `if-else`. You have loop statements `while` and `for`. You use indentation to indicate the scope of a selection or loop construct. You have an output statement `print`. And you have a `stop` statement. Although later in the class we'll see a few more parts of pseudocode, these are the essential parts and are enough to specify both simple and complicated algorithms.

## 2.5 Some Practice

This section contains some introductory practice problems. Answers are at the end of the chapter.

### 2.5.1 Writing Pseudocode Fragments

**Problem 4:** One stepping stone to writing an entire algorithm is to make sure you can write small fragments of pseudocode. Write pseudocode to do the following. In each case assume you have already gotten any input. Moreover, do not worry about printing any output.

- Set `count` to 0 if `a` is less than 5.
- Set `count` to 0 if `a` is less than 5 and `b` is less than 2; otherwise set `count` to 1.
- Add 1 to a five-digit odometer reading. Note a five-digit odometer can show values between 0 and 99999. Adding 1 to any odometer reading will simply increase that reading by 1 — for example 67817 will go to 67818 — except if the reading is 99999, in which case it will roll over to 0.
- Calculate the sum of the integers between 1 and 42.
- Count the number of T's in a genomic sequence (i.e., a sequence consisting of characters 'A', 'C', 'G', or 'T')  $A[1], \dots, A[n]$ .

- f. Count the number of times the two-character sequence ‘TG’ appears in a genomic sequence  $A[1], \dots, A[n]$ . That is, count the number of times a ‘T’ occurs in a location that is immediately followed by a ‘G’ in the next location.

### 2.5.2 Reading an Algorithm

**Problem 5:** Consider the following algorithm:

**Algorithm 4:**

*Input:* A nonempty string of characters  $S_1S_2 \dots S_n$ , and a positive integer  $n$  giving the number of characters in the string.

*Output:* See the related problem below.

```

1  Get  $n$ 
2  Get  $S_1S_2 \dots S_n$ 
3  Set count to 1
4  Set ch to  $S_1$ 
5  Set  $i$  to 2
6  While  $i \leq n$ 
7    If  $S_i$  equals ch
8      Set count to count + 1
9    Set  $i$  to  $i + 1$ 
10 Print ch, ‘ appeared ’, count, ‘ times.’
11 Stop
```

- What is printed if the input string is `pepper`?
- What is printed if the input string is `CACCTGGTCCAAC`?
- What is the output of this algorithm (in general)? Be precise.
- Suppose line 3 was changed to `Set count to 0`. How would your answer to part (c) change?

### 2.5.3 Common Mistakes

There are a numerous mistakes that can come up in writing pseudocode. Some are easier to make than others, and even experienced computer practitioners make them from time to time.

**Problem 6:** Suppose you are studying lake level data for a certain lake. This data reports the lake levels over a number of days, compared to the lake’s average (mean) level. Negative data correspond to times when the lake level was below average, positive to when it was above. So, for example, a lake level of  $-.23$  meters means that at the time of that reading the lake was  $.23$  meters lower than average. A reading of  $.12$  meters

means the level was .12 meters above the average level at the time of that reading. You would like to know how many readings were below the lake average.

More generally, the following pseudocode attempts to count the number of negative entries in an input list  $A[1], \dots, A[n]$ :

```
1 Get n
2 Get A[1], ..., A[n]
3 Set countNegativeEntries to 0
4 While i < n
5   If A[j] <= 0, then
6     Set countNegativeEntries to countNegativeEntries + 1
7   Print 'The number of negative entries is ', countNegativeEntries
8 Stop
```

Identify and correct any mistakes in this pseudocode.

## 2.6 Two Examples For Class

Here are two additional algorithm examples we will use in class. The first one performs a common operation, counting the number of times a given character occurs in a string (think of specific examples where this can be useful.) The second algorithm counts the number of times a given pattern (which can consist of more than one character) occurs in a string. This algorithm is more complicated, so expect to spend some time examining it.

**Algorithm 5: Character Count**

*Input:* A string length  $n$  and a text string  $S_1S_2 \dots S_n$  of alphabetic characters, as well as a search character  $ch$ .

*Output:* A message indicating how many times  $ch$  appears in  $S$ .

```

1  Get  $n$ 
2  Get  $S_1S_2 \dots S_n$ 
3  Get  $ch$ 
4  Set  $i$  to 1
5  Set  $count$  to 0
6  While  $i \leq n$  do
7      If  $S_i$  equals  $ch$  then
8          Set  $count$  to  $count + 1$ 
9      Set  $i$  to  $i + 1$ 
10 Print  $ch$ , ' appeared ',  $count$ , ' times.'
11 Stop
```

**Algorithm 6: Pattern Matching**

*Input:* A string length  $n$  and a text string  $S_1S_2 \dots S_n$  of alphabetic characters, as well as a pattern length  $m$  and a pattern  $P_1P_2 \dots P_m$  of alphabetic characters.

*Output:* A message indicating each time the pattern  $P$  matches a substring of  $S$ .

```

1  Get  $n$ 
2  Get  $S_1S_2 \dots S_n$ 
3  Get  $m$ 
4  Get  $P_1P_2 \dots P_m$ 
5  Set  $i$  to 1
6  While  $i \leq n - m + 1$  do
7      Set  $j$  to 1
8      Set  $matchOK$  to true
9      While  $j \leq m$  and  $matchOK$  equals true then
10         If  $P_j \neq S_{i+j-1}$  then
11             Set  $matchOK$  to false
12         Set  $j$  to  $j + 1$ 
13     If  $matchOK$  equals true then
14         Print 'Match found between positions ',  $i$ , ' and ',  $i + m - 1$ 
15     Set  $i$  to  $i + 1$ 
16 Stop
```

## 2.7 Writing Algorithms: How to Begin?

When you are writing an algorithm from scratch, often the most difficult part is getting started. In class we will discuss techniques — such as top-down design, working a concrete

example, or figuring out what you need to keep track of during an algorithm — that can be helpful. Here we'll give an example of one of these techniques, top-down design.

Suppose you are given a maze<sup>8</sup> that consists of a  $10 \times 10$  grid. That is, the maze consists of 10 rows and 10 columns of squares. Suppose that you can specify a row and column number, and do a check `isOpenNorth`, that returns `true` if there is an opening on the “north” side of the current square, and `false` otherwise. Also, you can do checks `isOpenEast`, `isOpenWest`, and `isOpenSouth` to check the east, west, and south sides, respectively.

Suppose further that, instead of solving the maze, you need to write an algorithm to go though and count the number of maze locations that are enclosed on exactly three sides (that is, the locations where there are walls on any three of the sides, but an opening on the fourth). To do this you are allowed to step through the maze as if it were a table, i.e., row by row and column by column. Put another way, you don't need to follow paths through the maze to visit each square; instead you have a “birds-eye view.”

What are the major tasks such an algorithm would need to do? A very crude high level outline is

```
Get input
Count locations enclosed on exactly 3 sides
Output result
```

Of these three items, the second will be the most complicated, so let's focus on it and break it down further. We need to step through all the locations and count the number of walls. So a more detailed breakdown can be something like this:

```
Step through each location
  If the current location has exactly 3 walls
    Count it
```

Note that this isn't pseudocode yet, though. For example, what does it mean to step through each location? What does it mean to count it? So we can be more specific:

```
For each row r
  For each column c
    If the location at row r column c is enclosed on exactly 3 sides
      Add one to the count
```

We're closer. But there are still additional details we need to include. For example, `For each row r` is still imprecise, as is `For each column c`. Another issue is we need to initialize count, and update it when appropriate. Here's another attempt:

---

<sup>8</sup>Important: If there is a maze problem on the homework, some parts of the notes will be helpful on that problem, but other parts will not. Do not rely overmuch on this example in solving any homework problem involving mazes.

```

Set r to 1
Set c to 1
Set count to 0
While r <= 10
  While c <= 10
    If the location at row r column c is enclosed on exactly 3 sides
      Set count to count + 1
    Set c to c + 1
  Set r to r + 1

```

To complete this part, notice that the condition `If the location is enclosed on exactly 3 sides` still needs to be made more specific as well.

**Problem 7:** Make the following line more specific:

```

    If the location at row r column c is enclosed on exactly 3 sides

```

## 2.8 Questions to Think About For Class

Here are questions to think about for class.

1. What types of problems can be solved by algorithms? What types can't, or can't easily?
2. Think about an area you are interested in outside of computer science. Are there any processes in that area that need to be carefully and precisely specified (regardless of whether they are done by computer or not)?
3. Cooking recipes, and instructions for putting together furniture are two analogies used for algorithms. For each one, explain how it is like an algorithm, and how it is not. Also, think of other analogies.
4. Suppose you are explaining algorithms to another student. What is hard about algorithms — what would you be sure to explain very carefully?
5. Come up with some specific examples where you might want the following: (a) a triply nested loop, (b) a loop inside an `if` statement, (c) one `if` statement inside another.

## 2.9 Additional Practice

Here are three additional practice problems. Solutions are in the next section.

**Problem 8:** Trace through Algorithm 6, the pattern matching algorithm, when the text string  $S$  is `CGCCCTACCGGCACC` and the pattern string  $P$  is `CC`. Specifically, (a) state what

the output would be, and (b) each time line 15 is reached write the current values of  $i$  (before 1 is added to  $i$  in that line),  $j$ , and  $matchOK$ .

**Problem 9:** This question again asks you to consider the pattern matching algorithm, Algorithm 6. Suppose that instead of printing a message whenever there was a match, you just want the algorithm to output the number of times matches occur. For example, suppose the text string  $S$  is ATGCATAGATT, and the pattern  $P$  is AT. Then the algorithm should output only the message

There were 3 matches.

Modify the pattern-matching algorithm to do this. For your answer you may either write the entire algorithm, or you may just indicate *exactly what* needs to be added or modified *exactly where*.

## 2.10 Problem Solutions

### 2.10.1 Introductory Problem

Here is a model solution to the problem at the beginning of this chapter. Note that for some problems, including this one, there is more than one possible correct solution. So this is one possible algorithm, but not the only possible correct one.

```

1  Get the input maze
2  startInMaze
3  moveForward
4  Set count to 1
5  While checkForMazeEnd returns false
6      If checkForWall returns true
7          turnRight
8          If checkForWall returns true
9              turnLeft          // turn back to original heading
10             turnLeft         // now turn left from that heading
11         moveForward
12         Set count to count + 1
13     Print 'Robot found end of maze.'
14     Print 'Number of moves: ', count
15     Stop

```

### 2.10.2 Additional Problems

**Problem 1:** This problem will likely be discussed in class or lab.

**Problem 2:** It is not a valid algorithm. It is neither correct nor finite since the  $i$  value is always 2 (the algorithm should contain a line **Set  $i$  to  $i + 1$**  between Lines 7 and 8) indented at the same level as the **if** in Line 6). Also, it fails to specify the output explicitly.

Here is a correct version:

**(Corrected) Algorithm 2**

*Input:* A total number of days  $n$ , and a list  $A$ , with  $A[i]$  giving the maximum temperature, in degrees Fahrenheit, on day  $i$ .

*Output:* A message stating the maximum temperature.

```

1  Get n
2  Get A[1], ..., A[n]
3  Set maxTemp to A[1]
4  Set i to 2
5  While i <= n
6      If A[i] > maxTemp
7          Set maxTemp to A[i]
8      Set i to i + 1
9  Print 'The maximum temperature was ', maxTemp
10 Stop

```

**Problem 3:** The pseudocode would print out the character 'y'. Note in line 3, **a** and **b** do not have equal values, since **a** has value 'x' and **b** has value 'y'. So the **else** statement is executed, setting the value of **a** to the value of **b**, namely 'y'.

**Problem 4:**

[a.]

```

    If a < 5
        Set count to 0

```

[b.]

```

    If a < 5 and b < 2
        Set count to 0
    Else
        Set count to 1

```

[c.]

```

    If odometerReading equals 99999
        Set odometerReading to 0
    Else
        Set odometerReading to odometerReading + 1

```

[d.]

```
Set i to 1
Set sum to 0
While i <= 42
    Set sum to sum + i
    Set i to i + 1
```

[e.]

```
Set i to 1
Set countT to 0
While i <= n
    If A[i] equals 'T'
        Set countT to countT + 1
    Set i to i + 1
```

[f.]

```
Set i to 1
Set countTG to 0
While i <= n - 1
    If A[i] equals 'T' and A[i+1] equals 'G'
        Set countTG to countTG + 1
    Set i to i + 1
```

**Problem 5:**

- a. p appeared 3 times.
- b. C appeared 6 times.
- c. A message stating the number of times the first character of the word appears in the word.
- d. The number in the output message would be the number of times the first character of the word appears in the word, excluding the first occurrence.

**Problem 6:** There are six errors.

1. `i` is never initialized — there should be a line `Set i to 1` prior to the while loop.
2. The loop continuation condition is incorrect: `i < n` stops before the last item in the list, `A[n]`. The loop condition should be `i <= n`.
3. In line 5 `A[j]` should be `A[i]`.

4. In line 5 `<=` should be `<`.
5. `i` is never updated in the loop: there should be a line `Set i to i + 1` at the end of the while loop.
6. The `print` line's indentation is incorrect: In the pseudocode the incorrect indentation places the `print` line inside the while loop, meaning it is executed during each iteration of that loop. It should be executed only once, after the while loop is done. Therefore its indentation should be the same as the 'While' in Line 4.

**Problem 7:**

```
Set countOpenings to 0
If isOpenNorth at row r column c equals true
    Set countOpenings to countOpenings + 1
If isOpenEast at row r column c equals true
    Set countOpenings to countOpenings + 1
If isOpenSouth at row r column c equals true
    Set countOpenings to countOpenings + 1
If isOpenWest at row r column c equals true
    Set countOpenings to countOpenings + 1
If countOpenings equals 3
    Set count to count + 1
```

**Problem 8:**

- a. The output would be

```
Match found between positions 3 and 4
Match found between positions 4 and 5
Match found between positions 8 and 9
Match found between positions 14 and 15
```

- b. Here are the values immediately prior to each execution of the line 15:

i	j	matchOK
---	---	-----
1	3	false
2	2	false
3	3	true
4	3	true
5	3	false
6	2	false
7	2	false
8	3	true
9	3	false
10	2	false
11	2	false
12	3	false
13	2	false
14	3	true

**Problem 9:** Here is one possibility.

*Input:* A string length  $n$  and a text string  $S_1S_2 \dots S_n$  of alphabetic characters, as well as a string length  $m$  and a pattern  $P_1P_2 \dots P_m$  of alphabetic characters.

*Output:* A message indicating the number of times the pattern  $P$  matches a substring of  $S$ . (changed)

```

1  Get  $n$ 
2  Get  $S_1S_2 \dots S_n$ 
3  Get  $m$ 
4  Get  $P_1P_2 \dots P_m$ 
5  Set  $i$  to 1
6  Set  $count$  to 0 // Added
7  While  $i \leq n - m + 1$  do
8    Set  $j$  to 1
9    Set  $matchOK$  to true
10   While  $j \leq m$  and  $matchOK$  equals true then
11     If  $P_j \neq S_{i+j-1}$  then
12       Set  $matchOK$  to false
13     Set  $j$  to  $j + 1$ 
14   If  $matchOK$  equals true then
15     Set  $count$  to  $count + 1$  // Changed
16   Set  $i$  to  $i + 1$ 
17 Print 'There were ',  $count$ , ' matches.' //Added
18 Stop
```

# Chapter 3

## Data Representation

It is all 0's and 1's.

### 3.1 Introductory Problem

Computers often represent colors as an RGB (red-green-blue) triple of numbers, where each of the red, green, and blue components is an integer between 0 and 255. For example, the color (255, 0, 10) has full red, no green, and a small amount of blue. Write an algorithm that takes as input the RGB components for a color, and returns a message indicating the largest component or components. For example, if the input color is (100, 255, 0), the algorithm should output “Largest component(s): green”. And if the input color is (255, 255, 255), then the algorithm should output “Largest component(s): red, green, blue”.

### 3.2 Overview

One amazing aspect of computers is they can store so many different types of data. Of course computers can store numbers. But unlike simple calculators they can also store text; and they can store colors, and images, and audio, and video, and many other types of data. And not only can they store many different types, but they can also analyze them, and they can transmit them to other computers. This versatility is one reason why computers are so useful, and affect so many areas of our lives.

To understand computers and computer science, it is important to know something about how computers deal with different types of data. Let's return to colors. How are colors stored in a computer? The introductory problem states one way, namely as an RGB triple. This is not the only possible way: RGB is just one of many color systems. For example, sometimes colors are represented as an HSV triple: by hue, saturation, and value. However, RGB is the most common color representation in computer programs.

This leads to a deeper issue: how are *numbers* stored in a computer? And why is it

important anyway that we understand how numbers, and other different types of data, are stored and processed in a computer? This chapter deals with these and related questions. In particular, we will look at the following:

1. Why is this an important topic?
2. How do computers represent numbers?
3. How do computers represent text?
4. How do computers represent other types of data such as images?
5. What is the binary number system and why is it important in computer science?
6. How do computers do basic operations such as addition and subtraction?

### 3.2.1 Goals

Upon completing this chapter, you should be able to do the following:

1. Be able to explain how, on the lowest level, computers represent both numeric and text data, as well as other types of data such as color data.
2. Be able to explain and use the basic terminology in this area: bit, byte, megabyte, RGB triple, ASCII, etc.
3. Be able to convert numbers and text from one representation to another.
4. Be able to convert integers from one representation to another, for example from decimal representation to two's complement representation.
5. Be able to add and subtract numbers written in unsigned binary or in two's complement representation.
6. Be able to explain how the number of bits used to represent data affects the range and precision of the representation.
7. Be able to explain in general how computers represent different types of data such as images.
8. Be able to do calculations involving amounts of memory or download times for certain datasets.

### 3.2.2 Data Representation and Mathematics

How is data representation related to the liberal education mathematics requirement? As you might guess, there is a strong connection. Computers store all data in terms of binary (i.e., base 2) numbers. So to understand computers it is necessary to understand binary. Moreover, you need to understand not only binary basics, but also some of the complications such as the “two’s complement” notation discussed below.

Binary representation is important not only because it is how computers represent data, but also because so much of computers and computing is based on it. For example, we will see it again in the chapter on machine organization.

### 3.2.3 Data Representation and Society and Technology

*The computer revolution.* That is a phrase you often hear used to describe the many ways computers are affecting our lives. Another phrase you might hear is the *digital revolution*. What does the digital revolution mean?

Nowadays, many of our devices are digital. We have digital watches, digital phones, digital radio, digital TVs, etc. However, previously many devices were *analog*. According to the Merriam-Webster online dictionary<sup>1</sup> “analog” means “of or relating to a device or process in which data is represented by physical quantities that change continuously.” Think, for example, of an old watch with second, minute, and hour hands that moved continuously (although very slowly for the minute and hour hands). Compare this with many modern-day watches that shows a digital representation of the time such as 2:03:23.

This example highlights a key difference between analog and digital devices: analog devices rely on a continuous phenomenon and digital on a discrete one. As a second example of this difference, an analog radio receives audio radio broadcast signals which are transmitted as radio *waves*, while a digital radio receives signals which are streams of numbers.<sup>2</sup>

The digital revolution refers to the many digital devices, their uses, and their effects. These devices include not only computers, but also other devices or systems that play a major role in our lives, such as communication systems.

Because digital devices usually store numbers using the binary number system, a major theme in this chapter is binary representation of data. Binary is fundamental to computers and computer science: to understand how computers work, and how computer scientists think, you need to understand binary. The first part of this chapter therefore covers binary basics. The second part then builds on the first and explains how computers store different types of data.

---

<sup>1</sup><http://www.merriam-webster.com/dictionary/>. Accessed Oct. 1, 2013.

<sup>2</sup>Actually, it’s more complicated than that because some devices, including some digital radios, intermix digital and analog. For example, a digital radio broadcast might start in digital form, i.e., as a stream of numbers, then be converted into and transmitted as radio waves, then received and converted back into digital form. Technically speaking the signal was *modulated* and *demodulated*. If you have a *modem* (*modulator-demodulator*) on your computer it fulfills a similar function.

## 3.3 Representation Basics

### 3.3.1 Introduction

The algorithms chapter discussed ways to describe a sequence of operations. Computer scientists use algorithms to specify *behavior* of computers. But for these algorithms to be useful they need data, and so computers need ways to represent data.<sup>3</sup>

People have many ways to represent even a very simple number. For example, the number four can be represented as 4 or IV or |||| or  $2 + 2$ , etc. How do computers represent numbers? (Or text? Or audio files?)

The way computers represent and work with numbers is different from how we do. Since early computer history the standard has been the binary number system. Computers “like” binary because it is extremely easy for them. However, binary is not easy for humans. Most of the time people do not need to be concerned with the internal representations that computers use; however, sometimes they do.

### 3.3.2 Why Binary?

Suppose you and some friends are spending the weekend at a cabin. The group will travel in two separate cars, and you all agree that the first group to arrive will leave the front light on to make it easier for the later group. When the car you are in arrives at the cabin you will be able to tell by the light if your car arrived first. The light therefore encodes two possibilities: on (the other group has already arrived) or off (the other group hasn’t arrived yet).

To convey more information you could use two lights. For example, both off could mean the first group hasn’t arrived yet, the first light off and second on indicate the first group has arrived but left to get supplies, the first on and second off that the group arrived but left to go fishing, and both on that the group has arrived and hasn’t left.

Note the key ideas here: a light can be on or off (we don’t allow different level of light, multiple colors, etc.), just two possibilities. But the second is that if we want to represent more than two choices we can use more lights.

This on or off idea is a powerful one. There are two and only two distinct choices or states: on or off, 0 or 1, black or white, present or absent, large or small, rough or smooth, etc. — all of these are different ways of representing possibilities. One reason the two-choice idea is so powerful is it is easier to build objects — computers, cameras, CDs, etc. — where the data at the lowest level is in two possible states, either a 0 or a 1.<sup>4</sup>

---

<sup>3</sup>Actually we need not only data, but a way to represent the algorithms within the computer as well. How computers store algorithm instructions is discussed in a later chapter.

<sup>4</sup>Of course how a 0 or 1 is represented varies according to the device. For example, in a computer the common way to differentiate a 0 from a 1 is by electrical properties, such as using different voltage levels. In a fiber optic cable, the presence or absence of a light pulse can differentiate 0’s from 1’s. Optical storage devices can differentiate 0’s and 1’s by the presence or absence of small “dents” that affect the

In computer representation, a *bit* (i.e., a binary digit) can be a 0 or a 1. A collection of bits is called a *bitstring*. A bitstring that is 8 bits long is called a *byte*. Bits and bytes are important concepts in computer storage and data transmission, and later on we'll explain them further along with some related terminology and concepts. But first let's look at the basic question of how a computer represents numbers.

### 3.3.3 Review of the Decimal Number System

We all know decimal (i.e., base 10) representation and use it every day. So, for example, the number one hundred and twenty-four is  $1 \times 100 + 2 \times 10 + 4 \times 1$ . We can emphasize this by writing the powers of 10 over the digits in 124:

$$\begin{array}{ccc} 10^2 & 10^1 & 10^0 \\ 1 & 2 & 4 \end{array}$$

So if we take what we know about base 10 and apply it to base 2 we can figure out binary. But first recall that a bit is a binary digit and a byte is 8 bits. In this file most of the binary numbers we talk about will be one byte long.

(Computers actually use more than one byte to represent most numbers. For example, most numbers are actually represented using 32 bits (4 bytes) or 64 bits (8 bytes). The more bits, the more different values you can represent: a single bit permits 2 values, 2 bits give 4 values, 3 bits gives 8 values, ..., 8 bits give 256 values, and in general  $n$  bits gives  $2^n$  values. When looking at binary examples we'll usually use 8 bit numbers to make the examples manageable.)

### 3.3.4 Unsigned Binary

When we talk about decimal, we deal with 10 digits — 0 through 9 (that's where *decimal* comes from). In binary we only have two digits, that's why it's *binary*. The digits in binary are 0 and 1. You will never see any 2's or 3's, etc. If you do, something is wrong. A bit will always be a 0 or 1.

Counting in binary proceeds as follows:

```

0      (decimal 0)
1      (decimal 1)
10     (decimal 2)
11     (decimal 3)
100    (decimal 4)
101    (decimal 5)
...

```

---

reflectivity of locations on the disk surface.

(Old joke: “There are 10 types of people in the world. Those who understand binary and those who don’t.”)

The next thing to think about is what values are possible in one byte. Let’s write out the powers of two in a byte:

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
128	64	32	16	8	4	2	1

As an example, the binary number 10011001 is

$$1 \times 128 + 0 \times 64 + 0 \times 32 + 1 \times 16 + 1 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1 = 153.$$

Note each of the 8 bits can either be a 0 or a 1. So there are two possibilities for the leftmost bit, two for the next bit, two for the bit after that, and so on: two choices for each of the 8 bits. Multiplying these possibilities together gives  $2^8$  or 256 possibilities. In *unsigned binary* these possibilities represent the integers between 0 (all bits 0) to 255 (all bits 1).

### 3.3.5 Decimal to Binary Conversion

One task you will need to do in this class, and which computer scientists often need to do, is to convert a decimal number to or from a binary number. The last subsection showed how to convert binary to decimal: take each power of 2 whose corresponding bit is a 1, and add those powers together.

Suppose we want to do a decimal to binary conversion. As an example, let’s convert the decimal value 75 to binary. Here’s one technique that relies on successive division by 2:

75/2	quotient=37	remainder=1
37/2	quotient=18	remainder=1
18/2	quotient=9	remainder=0
9/2	quotient=4	remainder=1
4/2	quotient=2	remainder=0
2/2	quotient=1	remainder=0
1/2	quotient=0	remainder=1

We then take the remainders bottom-to-top to get 1001011. Since we usually work with group of 8 bits, if it doesn’t fill all eight bits, we add zeroes at the front until it does. So we end up with 01001011.

**Problem 1:** Write an algorithm that specifies the process given in the example above to convert a decimal integer to binary. Here is the input and output specification:

*Input:* a nonnegative positive integer  $n$ .

*Output:* a list of digits  $b_k, b_{k-1}, \dots, b_1, b_0$  where  $b_0$  is the 1’s digit,  $b_1$  the 2’s digit, and  $b_k$  the largest (i.e., leftmost digit) in the binary representation of  $n$  (note we aren’t adding any 0’s to the front to get a predetermined length.)

The algorithm before Problem 1 is one common method for decimal to binary conversion. Here is another. Let's convert the decimal value 87 to binary. We start by finding the largest power of two that is not greater than 87. It is 64. We then put a '1' in the 64 (i.e., the  $2^6$ ) place in the binary representation of 87, and next subtract 64 from 87 to get 23. Now the next power of 2 down from 64 is 32. Since 23 is less than 32 we put a '0' in the 32 place. The next power down is 16. Since 23 is greater than 16, we put a '1' in the 16 place, subtract 16 from 23 to get 7, and continue with the process. Here is a short write-up of the remaining steps.

The next power of 2 downward is 8.

Is 7 greater than or equal to 8?

No, so put a 0 in the 8 place.

The next power of 2 downward is 4.

Is 7 greater than or equal to 4?

Yes, so put a 1 in the 4 place, and subtract 4 from 7 to get 3.

The next power of 2 downward is 2.

Is 3 greater than or equal to 2?

Yes, so put a 1 in the 2 place, and subtract 2 from 3 to get 1.

The next power of 2 downward is 1.

Is 1 greater than or equal to 1?

Yes, so put a 1 in the 1 place, and subtract 1 from 1 to get 0.

Since we have considered all powers of 2 down to 2 to the 0th power, namely 1, we stop.

Here is the representation of 87, written as a byte with the powers of two written above each bit:

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
128	64	32	16	8	4	2	1
0	1	0	1	0	1	1	1

Either of the two techniques above will work for converting decimal to binary.

**Tip:** Memorize the first 10 or so powers of 2. You'll be using them extensively in this class.

**Problem 2:** In one episode of the TV show *The Simpsons* the character Homer Simpson wrote the following:

$$3987^{12} + 4365^{12} = 4472^{12}.$$

(Leave aside for now the question of whether this is a correct equation.) Represent each of the three numbers 3987, 4365, and 4472 in binary. Use as many bits as needed for each number (you will need more than eight).

### 3.3.6 Addition of Binary Numbers

In addition to storing data, computers also need to do operations such as addition of data. How do we add numbers in binary representation?

Addition of bits has four simple rules:

$$\begin{array}{cccc}
 0 & 0 & 1 & 1 \\
 +0 & +1 & +0 & +1 \\
 \hline
 0 & 1 & 1 & 10
 \end{array}$$

Now if we have a binary number consisting of multiple bits we use these four rules, plus “carrying”. Here’s an example:

$$\begin{array}{r}
 00110101 \\
 + 10101100 \\
 \hline
 11100001
 \end{array}$$

Here’s the same example, but with the carried bits listed explicitly, i.e., a 0 if there is no carry, and a 1 if there is:

$$\begin{array}{r}
 \text{carry :} \quad 0111100 \\
 \quad \quad \quad 00110101 \\
 \quad \quad \quad + 10101100 \\
 \hline
 \quad \quad \quad 11100001
 \end{array}$$

We can check binary operations by converting each number to decimal: with both binary and decimal we’re doing the same operations on the same numbers, but with different representations. If the representations and operations are correct the results should be consistent. Converting 00110101 to decimal produces 53 (do the conversion on your own to verify its accuracy), and converting 10101100 gives 172. Adding these yields 225, which, when converted back to binary is indeed 11100001.

But ... binary addition of two 8-bit numbers doesn’t always work quite right:

$$\begin{array}{r}
 01110100 \\
 + 10011111 \\
 \hline
 100010011
 \end{array}$$

Note there are 9 bits in the result, but there should only be 8 in a byte. Here is the sum in decimal:

$$\begin{array}{r}
 116 \\
 + 159 \\
 \hline
 275
 \end{array}$$

Note 275 is greater than 255, which is the maximum we can hold in an 8-bit number. This results in a condition called *overflow*. Overflow is not an issue if the computer can go to a 9-bit binary number; however, if the computer only has 8 bits set aside for the result, overflow can result in the program not running, or not running correctly.

### 3.3.7 Subtraction of Binary Numbers

Once again, let's start by looking at single bits:

$$\begin{array}{r} 0 \quad 0 \quad 1 \quad 1 \\ -0 \quad -1 \quad -0 \quad -1 \\ \hline 0 \quad -1 \quad 1 \quad 0 \end{array}$$

Notice that in the  $-1$  case what we often want to do is get a bit value 1, and borrow. So let's apply this to an 8-bit problem:

$$\begin{array}{r} 10011101 \\ - 00100010 \\ \hline 01111011 \end{array}$$

which is the same as (in base 10)

$$\begin{array}{r} 157 \\ - 34 \\ \hline 123 \end{array}$$

Here's the binary subtraction again with the borrowing shown:

$$\begin{array}{r} \text{borrow :} \quad 1100010 \\ \quad 10011101 \\ - \quad 00100010 \\ \hline \quad 01111011 \end{array}$$

Most people find binary subtraction significantly harder than binary addition.

**Problem 3:** In the last subsection we saw that overflow was a possible problem when two binary numbers were added. Can it (or a similar condition) occur when one binary number is subtracted from another?

## 3.4 Other Representations Related to Binary

You might have had questions about the binary representation in the last section. For example, what about negative numbers? What about numbers with a fractional part?

Aren't all those 0's and 1's difficult for humans to work with? Etc. These are good questions. In this and the next two sections we'll look at a few other representations that are used in computer science and are related to binary.

### 3.4.1 Hexadecimal

Computers are good at binary. Humans aren't. Binary is hard for humans to write, hard to read, and hard to understand. But what if we want a number system that is easier to read, etc. but still is closely tied to binary in some way?

One possibility is *hexadecimal*, i.e., base 16. But using a base greater than 10 immediately presents a problem. Specifically, we run out of digits after 0 to 9 — we can't use 10, 11, etc. because those have multiple digits within them. So instead we use letters: A is 10, B 11, C 12, D 13, E 14, and F 15, as shown in Figure 3.1. So the digits we're using are 0 through F instead of the 0 through 9 in decimal, or the 0 and 1 in binary.

Figure 3.1: Hexadecimal digits and their decimal equivalents

Hexadecimal Digit	Decimal Equivalent
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

We also have to reexamine the value of each place. In hexadecimal, each place represents a power of 16. A two-digit hexadecimal number has a 16's place and a 1's place.

For example, D8 has D in the 16's place, and 8 in the 1's place:

$$\begin{array}{r} 16^1 \quad 16^0 \\ 16 \quad 1 \\ D \quad 8 \end{array}$$

So the hexadecimal number D8 equals  $13 \times 16 + 8 \times 1 = 216$  in decimal. Note any two digit hexadecimal number, however, can represent the same amount of information as one byte of binary. So it's easier for us to read or write.

When working with a number, there are times when which representation is being used isn't clear. For example, does 10 represent the number ten (so the representation is decimal), the number two (the representation is binary), the number sixteen (hexadecimal), or some other number? Often, the representation is clear from the context. However, when it isn't we use a subscript to clarify which representation is being used, for example  $10_{10}$  for decimal, versus  $10_2$  for binary, versus  $10_{16}$  for hexadecimal.

Hexadecimal numbers can have more hexadecimal digits than two. For example, consider *FF0581A4*, which uses the following powers of 16:

$$\begin{array}{cccccccc} 16^7 & 16^6 & 16^5 & 16^4 & 16^3 & 16^2 & 16^1 & 16^0 \\ F & F & 0 & 5 & 8 & 1 & A & 4 \end{array}$$

So in decimal this is

$$\begin{aligned} & 15 \times 16^7 + 15 \times 16^6 + 0 \times 16^5 + 5 \times 16^4 + 8 \times 16^3 + 1 \times 16^2 + 10 \times 16^1 + 4 \times 16^0 \\ = & 15 \times 268435456 + 15 \times 16777216 + 0 \times 1048576 + 5 \times 65536 \\ & + 8 \times 4096 + 1 \times 256 + 10 \times 16 + 4 \times 1 \\ = & 4,278,550,948 \end{aligned}$$

**Problem 4:** How many bits are required to store the hexadecimal number *FF0581A4*?

Hexadecimal doesn't appear often, but it is used in some places, for example sometimes to represent memory addresses (you'll see this in a future chapter) or colors. Why is it useful in such cases? Consider a 24-bit RGB color with 8 bits each for red, green, and blue. Since 8 bits requires 2 hexadecimal digits, a 24-bit color needs 6 hexadecimal digits, rather than 24 bits. For example, *FF0088* indicates a 24-bit color with a full red component, no green, and a mid-level blue.

Now there are additional types of conversion problems:

- Decimal to hexadecimal
- Hexadecimal to decimal
- Binary to hexadecimal
- Hexadecimal to binary

Here are a couple examples involving the last two of these.

Let's convert the binary number 00111100 to hexadecimal. To do this, break it into two 4-bit parts: 0011 and 1100. Now convert each part to decimal and get 3 and 12. The 3 is a hexadecimal digit, but 12 isn't. Instead recall that C is the hexadecimal representation for 12. So the hexadecimal representation for 00111100 is 3C.

Rather than going from binary to decimal (for each 4-bit segment) and then to hexadecimal digits, you could go from binary to hexadecimal directly using Figure 3.2.

Figure 3.2: Hexadecimal digits and their decimal and binary equivalents

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Now let's convert the hexadecimal number D6 to binary. D is the hexadecimal representation for  $13_{10}$ , which is 1101 in binary. 6 in binary is 0110. Put these two parts together to get 11010110. Again we could skip the intermediate conversions by using the hexadecimal and binary columns in the Figure 3.2.

### 3.4.2 Sign/Magnitude Notation

Thus far we've been working with positive numbers. What about negatives? For example, suppose the temperature is  $-15^{\circ}$  F. How would we represent this in binary?

One possibility is to use one bit to indicate the sign of the number. Let's use the leftmost bit: instead of it being the 128's place we interpret it to indicate that the number is negative if that bit is a 1, and the number is positive if that bit is a 0. So, for

example, positive 39 is 00100111, but  $-39$  would be 10100111. We call this representation *sign/magnitude binary*, or just *sign/magnitude* for short.

This representation works, but only to some extent. Let's take a minute and look at the tradeoffs. First, with the unsigned binary representation in the last section, we can represent the integers 0 to 255 with a single byte. With the sign/magnitude representation we can still only represent at most 256 different possibilities. Note that the smallest number will be when all the bits are 1:

$$\begin{array}{rcccccccc} \text{sign} & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ +/\text{-} & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array}$$

This number is  $-(64 + 32 + 16 + 8 + 4 + 2 + 1) = -127$  in decimal.<sup>5</sup>

How about the largest number: This will be when all bits save the leftmost are 1:

$$\begin{array}{rcccccccc} \text{sign} & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ +/\text{-} & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ & 0 & 1 & 1 & 1 & 1 & 1 & 1 \end{array}$$

This is the same number as above, except  $+127$  instead of  $-127$ .

So by using this sign/magnitude notation we trade off representing more positive numbers for being able to represent some negative numbers.

Note that this representation allows 127 positive numbers, 127 negative numbers, and 0. This is 255 possibilities. You might remember that with unsigned binary a byte can represent 256 possibilities: the numbers 1 to 255, as well as 0. Where did the other possibility go with sign/magnitude representation?

Notice that 10000000 and 00000000 both represent 0 ( $-0$  and  $+0$ , respectively, which are both the same value, 0). This complicates matters. For example, suppose you are writing an accounting program and wanted the program to check if *revenue*  $-$  *expenses* equals 0. Doing this would require two checks, one against 10000000, and one against 00000000. This might not seem like a big deal, but it is just one of a number of complications that having two 0 representations introduces.

Let's look at another problem with sign/magnitude representation. Does addition work? For example, what happens if we add 30 to  $-39$ ? Will the usual way of doing binary addition work? We should get  $-9$ . Here's what we do get:

$$\begin{array}{r} 00011110 \\ + 10100111 \\ \hline 11000101 \end{array}$$

<sup>5</sup>Note that in this calculation  $64 + 32 + \dots + 2 + 1 = 2^7 - 1$ . In general  $2^m + 2^{m-1} + 2^{m-2} + \dots + 2^1 + 2^0 = 2^{m+1} - 1$ . This is a useful formula to remember.

We know this number is negative, but what is it? It's the negative of  $64+4+1$ , or  $-69$ . So instead of adding 30, we subtracted 30.

This is not an insurmountable problem; there are ways to fix it. However, we would like a representation to be as efficient as possible for frequently performed operations. Is there a better option than sign/magnitude binary? Is there a representation that allows both positive and negative numbers, but that is more efficient? We'll see such a representation in the next section.

## 3.5 Two's Complement Representation

*Two's complement* is another method for representing numbers in binary. It's hard to understand at first, but the key points are it allows representing both positive and negative numbers, and does so in such a way that it avoids the problems that arise with the sign/magnitude representation.

### 3.5.1 Two's Complement Representation Basics

Here is one way to think about two's complement representation. We can start at 0 and count up:

```
00000000
00000001
00000010
00000011
...
```

And we can start at 0 but count down. What do we get if we start at 00000000 and subtract one at a time? We get the sequence (ignore for now the carry that "falls off" the left end):

```
11111111
11111110
11111101
11111100
...
```

This gives some hint as to how two's complement works. Specifically, for normal unsigned binary we have the following powers of 2 for each bit:

```
128 64 32 16 8 4 2 1
```

For sign/magnitude notation we have leftmost bit indicating the sign:

```
-/+ 64 32 16 8 4 2 1
```

Two's complement representation uses a different leftmost bit. Specifically, rather than being the 128's place, or indicating the sign, the leftmost bit corresponds to *negative* 128:

-128 64 32 16 8 4 2 1

One effect of this is the numbers that a byte can represent in two's complement don't range from 0 to 255 (as in unsigned binary). Instead the smallest number is  $-128$  (in two's complement 10000000), and the largest is  $+127$  (in two's complement 01111111).

Let's look at how we do conversions in this new representation. Here's an outline of an algorithm for converting from decimal to two's complement:

*Input:* an integer (represented in decimal) between  $-128$  and  $127$

*Output:* the two's-complement representation of the number.

Convert the absolute value of the number to binary

If the number is negative

Complement the binary representation (change 0's to 1's, and 1's to 0's)

Add one to the binary representation

Notice that changing a positive integer (between 0 and 127) to two's-complement is the same as we've already learned — you get the same bits as with the unsigned binary representation because the  $-128$  bit will be 0. But let's look at an example of converting a negative number. Specifically, let's convert  $-99$  to two's complement.

First we convert 99 to binary: 01100011

Then complement the bits: 10011100

Then add one: 10011101

We can do a two's complement to decimal conversion to check our work:

$$10011101 = 1 \times (-128) + 1 \times 16 + 1 \times 8 + 1 \times 4 + 1 \times 1 = -128 + 16 + 8 + 4 + 1 = -99.$$

### 3.5.2 Addition and Subtraction

We can now do two's complement addition using the usual binary process. For example, consider  $(-99) + 42$ . Note the two's complement representation of 42 is 00101010, and of  $-99$  is 10011101. Adding these gives

$$\begin{array}{r} 10011101 \\ + 00101010 \\ \hline 11000111 \end{array}$$

Notice if you convert the result from two's complement to decimal you get  $1 \times (-128) + 1 \times 64 + 1 \times 4 + 1 \times 2 + 1 \times 1 = -57$ , which is correct.

Here are a couple more examples. First  $107 + (-67)$ . The two's complement representation of 107 is 01101011, and of  $-67$  is 10111101. Adding these gives

$$\begin{array}{r} 01101011 \\ + 10111101 \\ \hline 00101000 \end{array}$$

(You can ignore for now the carry that falls off the left end.)

Next  $(-27) + (-67)$ . The two's complement representation of  $-27$  is 11100101, and of  $-67$  is 10111101. Adding these gives

$$\begin{array}{r} 11100101 \\ + 10111101 \\ \hline 10100010 \end{array}$$

(Again, you can ignore for now the carry that falls off the left end.)

Overflow can still be a problem with two's complement; it's a problem any time you have a fixed number of bits. For example, if you add  $(-128) + (-128)$  you get the following:

$$\begin{array}{r} 10000000 \\ + 10000000 \\ \hline 00000000 \end{array}$$

with a 1 bit carried off the left end. This is not the correct answer. Or if you add  $64 + 64$  you get another wrong result

$$\begin{array}{r} 01000000 \\ + 01000000 \\ \hline 10000000 \end{array}$$

which is again incorrect.

So while two's complement has many advantages, computers do need to do special checks of the leftmost bit and any leftover carry bits to ensure there isn't any overflow. The rules on this are as follows:

1. If the leftmost bit of both numbers to be added are 0, and the result of adding the two numbers gives a 1 as the leftmost bit of the result, then (as in the example immediately above) overflow has occurred.
2. If the leftmost bit of both numbers to be added are 1, then the leftmost bit of the result (excluding any carry bit off the left end) is a 0, then overflow has occurred.
3. In all other cases the result is correct, and we ignore any carry bit that "falls off" the left end of the result.

One final reminder on two's complement: It presents an elegant and efficient way of dealing with subtraction, namely by turning a subtraction problem into an addition one. We actually already used this observation in one of the examples above, since  $107 + (-67)$  is the same as  $107 - 67$ . But here is another example.

Suppose we want to do the subtraction  $81 - 27$  using two's complement. Note that in two's complement 81 is 01010001 and 27 is 00011011. But rather than subtracting these directly, we can write  $81 - 27$  as an addition problem, namely  $81 + (-27)$ . So instead of working with the representation of 27, we use the two's complement representation of  $-27$ , which we find by complementing 00011011 (change 0's to 1's and vice versa) and then adding 1. That is, we do the following:

00011011 start with the binary representation of 27  
 11100100 flip all the bits  
 11100101 add 1 to get the two's complement representation of  $-27$

Now do the addition:

$$\begin{array}{r} 01010001 \\ + 11100101 \\ \hline 00110110 \end{array}$$

Note that because the leftmost bits of both numbers being added are a 0 and a 1, overflow does not occur. And we ignore the carry bit that falls off the left edge. Moreover, the result checks:  $0 \times (-128) + 0 \times 64 + 1 \times 32 + 1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 = 54$ , which is correct.

## 3.6 Range

Before looking at how computers store other types of data, let's explore *range* further. The range of a representation is the set of numbers it can represent.

Up to this point we've been working primarily with bytes, or 8-bit bitstrings. Most computers and computer programs use more bits to represent numbers. For example, it's common to use 16 bits (2 bytes), or 32 bits (4 bytes), or even 64 bits to represent integers.

Let's examine how many more possibilities these additional bits provide. Consider the following questions:

1. How many different possibilities are there with 16 bits?
2. How many different possibilities are there with 32 bits?
3. If you use an unsigned binary representation with 16 bits, what decimal values can be represented?
4. If you use a two's complement representation with 32 bits, what decimal values can be represented?

To answer the first and second question, remember that each bit has two possibilities: 0 or 1. So if you have one bit, you have two possibilities. If you have two bits, you have four possibilities. If three, then eight. If four, then 16. And in general, if you have  $n$  bits, you have  $2^n$  possibilities. So 16 bits provide  $2^{16} = 65,536$  possibilities. 32 bits provide  $2^{32} = 4,294,967,296$  (over 4 billion) possibilities. (And 64 bits provide 18,446,744,073,709,551,616.)

Now examine a 16-bit unsigned binary representation. As we just saw, you have 65,536 possibilities. However, we include 0 as one of these, so the maximum decimal number in this representation is 65,535.

Next, consider a 32-bit two's complement representation. The largest number in this representation has all bits save the leftmost being 1. This has value  $2^{30} + 2^{29} + \dots + 2^1 + 2^0 = 2^{31} - 1 = 2,147,483,647$ . The smallest (i.e., most negative) number in this representation has leftmost bit 1 and all other bits 0. This has value  $-2^{31} = -2,147,483,648$ . So for 32-bit two's complement, the range is all integers between  $-2,147,483,648$  and  $2,147,483,647$ , inclusive.

Note that the more bits a representation has, the larger its range. Therefore, it might seem like a good strategy to always use an extremely large number of bits. However, there is a tradeoff: the more bits, the more space a computer needs to store a number, the less efficient the program will be. If, for example, you are storing a large number of colors, and the colors are in RGB color representation with 8 bits each for the red, green, and blue color components, then there is no need to use a 32-bit two's complement representation for each component.

Often programmers don't know in advance how large the range needs to be. For example, companies grow, and a small company that has tens of thousands of dollars of yearly revenue currently might have millions of dollars of yearly revenue in a decade. In general, programs try to provide a large enough range that they can store all numbers that will reasonably occur in the running of the program. If they provide too large of a range, they are not as efficient as they could be. If they provide too small of a range, the program will not be able to handle all numbers that arise — for example, overflow might occur.

Suppose you are keeping track of a company's yearly revenue in a computer program. You don't know in advance how large these numbers might be, but if you are familiar with the company you could estimate the needed range. But suppose your estimate is incorrect. Can't a computer just automatically use extra bits to get a larger range?

The answer to this question is “yes, but ...”. It is certainly possible for computers to do this. However, it is more complicated, so programs or computer languages that allow varying representation sizes are less efficient than those where items have a fixed length. For this reason, many computer languages require items to have a fixed length during program execution. However, some other languages allow the representation size to change. Once again, it is a matter of tradeoffs.

## 3.7 Floating Point Numbers

Of course not all numbers are integers. A gallon of milk might cost \$3.75. The distance in miles of a 10K race is (to the nearest tenth of a mile) 6.2. The number of days in a year is about 365.24 (which is the reason for leap years). Computers call such numbers, where there is a fractional part or digits to the right of the decimal point *floating point numbers*. We've seen how computers store integers. How do they represent floating point numbers?

Floating point representation is interesting, but also complicated. Rather than going into all the details here, we will just make a few key, high-level observations:

1. *Many of the issues with integer representation also occur with floating point numbers.* For example, there is again a trade-off between how many possible numbers you can represent, and how many bits you use.
2. *Floating point representations break a number into parts and include each part in the representation.* Specifically, recall scientific notation. For example, in scientific notation the number 32,000,000 would be represented as  $3.2 \times 10^7$ . When computers represent a floating point number, they represent it as the *mantissa*, which is 3.2 in the example above, and the *exponent*, which is 7.

(Note that the mantissa above is still not an integer. However, just as we can represent the price of an object as an integral number of cents rather than dollars, for example 375 cents instead of \$3.75, so we can also adjust a fixed width floating point number and represent it as an integer, as long as we know where the decimal point was originally.)

3. *How many bits you use for the mantissa and exponent affects the representation's precision and range.*

Because we need to store both the mantissa and exponent, there needs to be a standard for how many bits to devote to each. The more bits for one, the larger number of possible values it can take on, but the fewer bits for the other. A larger number of bits for the mantissa will give more precision, that is more digits to the right of the decimal point. But a larger number of bits for the exponent will give a larger exponent range.

Computer engineers have devised a number of standards for this. For example, the IEEE 7540 standard for a 32-bit floating point number results in about 7 decimal digits of accuracy, and allows exponents for powers of 10 between about  $10^{-38}$  and  $10^{38}$ . The standard for a 64-bit number gives both more precision (15 or 16 decimal digits) and a larger exponent range (giving powers of 10 between about  $10^{-308}$  and  $10^{308}$ ).

4. *There are a number of additional complications with floating point numbers.* Obviously, adding numbers that have both a mantissa and an exponent will be sig-

nificantly more complicated than adding, say, two 8-bit two’s complement integers. Moreover, there are other complications such as the desirability of including a representation for infinity, as well as for “NaN” (not a number) to indicate the result of an operation such as division by 0. We won’t go into the details for this.

However, there are two additional complications that deserve mentioning. The first is that computers cannot represent all numbers. Just as you can’t write down all the digits in the decimal representation of  $1/3$ , since it is  $.3333\dots$  with the three’s going on forever, so a computer can’t exactly represent many numbers exactly. The second complication is that, in a case like this, the computer does the same thing we often do if we need to work with the decimal representation of  $1/3$ : it uses an approximation rather than the exact number. This means — since all fixed-length representations have a finite, set number of bits — that representing numbers in a computer can result in *round-off error*. Moreover, round-off error can accumulate over a sequence of operations. This raises questions of how accurate the results of numerical computations are. For example, if a company is using a computer to do a wind tunnel simulation of a new airplane wing design, should they trust the results? *Numerical analysis* and *numerical computation* are the branches of mathematics and computer science that deal with these and other computational issues.

## 3.8 Text

So far this chapter has covered only numbers. Numbers are certainly important, but much of our data and much of what we do on computers doesn’t involve numbers. Many people who use computers primarily work with text: typing papers, reports, emails, etc. How do computers store text, i.e., character data?

To work with text, a computer must turn it into numbers, i.e., into binary. This requires each text character having a numerical equivalent. Then the computer works with these numbers, and thereby works with text.

There are, of course, many possible ways to do this, and many associated questions. For instance, how many bits will we need? The Latin alphabet has 26 characters, so if we use a fixed bitstring length for all characters we need the smallest  $n$  so that  $2^n \geq 26$ , namely 5. But that’s not quite right — we need both upper and lower case, so 52 possibilities. But even that is not right, since we also need punctuation marks. Moreover, it would also be nice to be able to encode items such as a tab or return or escape: non-printing characters that can play a role in printed text or in computer operations.

*ASCII* is a commonly used encoding that uses 8 bits, i.e., one byte, per character. For example, here are some ASCII equivalents: the capital ‘A’ is represented by the decimal number 65 (equivalently, the binary 01000001), the capital ‘B’ by 66, lower case ‘a’ by 97, the semicolon by 59, the space by 32, the escape key by 27, and the digit 4 by 52.<sup>6</sup> The

---

<sup>6</sup>Note that ASCII needs to represent all these characters, including the digits 0 through 9. This can be a little confusing because we’ve just gotten through discussing how computers represent numbers in

entire ASCII table can be found online. For example, see <http://www.asciitable.com> or <http://www.ascii-table.com>.

Note that even though we are mostly discussing decimal equivalents here, within the computer those equivalents are stored in binary. This brings up an important principal: *a bitstring in and of itself has little meaning; it is only when you (or the computer) know what type of representation is being used that you (or it) can say what the bitstring represents.* For example, consider the bitstring 01000001 (which equals  $65_{10}$ ). We just saw that if we interpret it as ASCII, it represents the character ‘A’. If we interpret it as unsigned binary, it represents the (decimal) number 65. Or, as another example, 11000001 represents 193 in unsigned binary,  $-65$  in sign/magnitude binary, and  $-63$  in two’s complement.<sup>7</sup>

However, with only one byte per character, ASCII is limited. Specifically, what if you want additional characters beyond those given in ASCII? For example, what if you want to use the Greek alphabet?

UNICODE is a set of standards for representing text in non-Latin alphabets, such as Russian, Arabic, Balinese, and Chinese. It can represent over 65,000 symbols by giving each one a 16-bit representation. For example, a Greek lower case letter pi has unicode representation 003C in hexadecimal.<sup>8</sup> And the Old Coptic small letter dja has representation 2CD9. You can find the UNICODE character charts online, for example at <http://www.unicode.org>.

Let’s go back to working with ASCII. Suppose we’re working with a program using ASCII. How much space does it take to encode a page of text? Obviously that will depend on a number of factors including the page height and width, font size, and interline spacing. Consider this set of notes. There are roughly 40 lines per page, and roughly 80 characters per line. This gives 3200 characters. In ASCII each character takes a single byte, so this is a little over 3000 bytes. In terms of computer file sizes, this is not much, not at all.<sup>9</sup> When we get to audio and image files below, we will see they take much more space. This leads to another principle: *text and numbers are easy (to store, transmit, etc.); images, audio, etc. are hard.*

---

binary. Why does ASCII need to represent character digits differently? Suppose you needed to decide how to encode the page above. Note it contains not only alphabetic characters but also numbers. Which is easier: encoding the numbers one way and the text characters another, or encoding all the characters using the same system? ASCII makes the latter choice for reasons of efficiency. Computers have special mechanisms — such as declarations in computer programs and programming languages — to specify when numbers need to be represented directly in binary rather than in ASCII.

<sup>7</sup>11000001 also has a meaning in ASCII; however, there are different versions of ASCII for the decimal equivalents 128 to 255. For example, in the ISO Latin-1 extended ASCII encoding, 11000001 represents a capital ‘A’ with an acute punctuation symbol.

<sup>8</sup>Here we are following the identification convention at <http://www.unicode.org> and using hexadecimal since it is easier to read and write than 16-bit binary.

<sup>9</sup>If you use a word processing program, the program might also store additional information (which will take additional space) about page format, font size and type, etc. However, the overall file size will still be relatively small.

Figure 3.3: Some Important Memory Amounts

Memory Amount	Power of Two	Approximation
1 kilobyte (KB)	$2^{10}$ bytes	thousand bytes
1 megabyte (MB)	$2^{20}$ bytes	million bytes
1 gigabyte (GB)	$2^{30}$ bytes	billion bytes
1 terabyte (TB)	$2^{40}$ bytes	trillion bytes
1 petabyte (PB)	$2^{50}$ bytes	quadrillion bytes
1 exabyte (EB)	$2^{60}$ bytes	quintillion bytes

### 3.9 Bytes, Kilobytes, Megabytes, and More

In the last section we saw that a page of text could take a few thousand bytes to store. Images files might take tens of thousands, hundreds of thousands, or even more bytes. Music files can take millions of bytes. Movie files can take billions. There are databases that consist of trillions or quadrillions of bytes of data.

Computer science has special terminology and notation for large numbers of bytes, as shown in Figure 3.3. There are still higher numbers or smaller quantities of these types. See, for example, [http://en.wikipedia.org/wiki/Binary\\_prefix](http://en.wikipedia.org/wiki/Binary_prefix).

Kilobytes, megabytes, etc. are important enough for discussing file sizes, computer memory sizes, etc. that you should know both the terminology and the abbreviations. One caution: file sizes are usually given in terms of *bytes* (or kilobytes, megabytes, etc.). However, some quantities in computer science are usually given in terms involving bits. For example, download speeds are often given in terms of bits per second. “Mbps” is an abbreviation for *megabits* (not megabytes) per second. Notice the ‘b’ in Mbps is a lower case, while the ‘b’ in MB (megabytes) is capital.

In the context of computer memory, the usual definition of kilobytes, megabytes, etc. is a power of two. For example, a kilobyte is  $2^{10} = 1024$  bytes, not a thousand. In some other situations, however, a kilobyte is defined to be exactly a thousand bytes. This can obviously be confusing. For the purposes of this class, the difference will usually not matter. That is, in most problems we do, an approximation will be close enough. So, for example, if we do a calculation and find a file takes 6,536 bytes, then you can say this is approximately 6.5 KB, unless the problem statement says otherwise.<sup>10</sup>

---

<sup>10</sup>The difference between “round” numbers, such as a million, and powers of 2 is not as pronounced for smaller numbers of bytes as it is for larger. A kilobyte is  $2^{10} = 1024$  bytes, which is only 2.4% more than a thousand. A megabyte is  $2^{20} = 1,048,576$  bytes, about 4.9% more than one million. A gigabyte is about 7.4% bytes more than a billion, and a terabyte is about 10.0% more bytes than a trillion. In most of the file size problems we do, we’ll be interested in the approximate size, and being off by 2% or 5% or 10% won’t matter. But of course there are real-world applications where it does matter, so when doing file size problems keep in mind we are doing approximations, not exact calculations.

## 3.10 Image Files, Audio Files, and Video Files

Images, audio, and video are other types of data. How computers represent these types of data is fascinating but complex. For example, there are perceptual issues (e.g., what types of sounds can humans hear, and how does that affect how many numbers we need to store to reliably represent music?), size issues (as we'll see below, these types of data can result in large file sizes), standards issues (e.g., you might have heard of JPEG or GIF image formats), and other issues.

We won't be able to cover image, audio, and video representation in depth: the details are too complicated, and can get very sophisticated. For example, JPEG images can rely on an advanced mathematical technique called the discrete cosine transform. However, it is worth examining a few key high-level points about image, audio, and video files:

1. Computers can represent not only basic numeric and text data, but also data such as music, images, and video.
2. They do this by digitizing the data. At the lowest level the data is still represented in terms of bits, but there are higher-level representational constructs as well.
3. There are numerous ways to encode such data, and so standard encoding techniques are useful.
4. Audio, image, and video files can be large, which presents challenges in terms of storing, processing, and transmitting these files. For this reason most encoding techniques use some sophisticated types of compression.

### 3.10.1 Images

“The largest and most detailed photograph of our galaxy ever taken has been unveiled.

The gigantic nine-gigapixel image captures more than 84 million stars at the core of the Milky Way.

It was created with data gathered by the Visible and Infrared Survey Telescope for Astronomy (VISTA) at the European Southern Observatory's Paranal Observatory in Chile.

If it was printed with the resolution of a newspaper it would stretch 30 feet long and 23 feet tall, the team behind it said, and has a resolution of 108,200 by 81,500 pixels.<sup>11</sup>

While this galaxy image is obviously an extreme example, it illustrates that images (even much smaller images) can take significant computer space. Here is a more mundane

---

<sup>11</sup>From [http://www.huffingtonpost.co.uk/2012/10/25/largest-ever-photo-of-the-galaxy-vista\\_n\\_2014208.html](http://www.huffingtonpost.co.uk/2012/10/25/largest-ever-photo-of-the-galaxy-vista_n_2014208.html). Accessed Nov. 5, 2013.

example. Suppose you have an image that is 1500 pixels wide, and 1000 pixels high. Each pixel is stored as a 24-bit color. How many bytes does it take to store this image?

This problem describes a straightforward but naive way to store the image: for each row, for each column, store the 24-bit color at that location. The answer is  $(1500 \times 1000)$  pixels  $\times$  24 bits/pixel  $\times$  1 byte/8 bits = 4.5 million bytes, or about 4.5MB.

Note the file size. If you store a number of photographs or other images you know that images, and especially collections of images, can take up considerable storage space. You might also know that most images do not take 4.5MB. And you have probably heard of some image storage formats such as JPEG or GIF.

Why are most image sizes tens or hundreds of kilobytes rather than megabytes? Most images are stored not in a direct format, but using some compression technique. For example, suppose you have a night image where the entire top half of the image is black ((0,0,0) in RGB). Rather than storing (0,0,0) as many times as there are pixels in the upper half of the image, it is more efficient to use some “shorthand.” For example, rather than having a file that has thousands of 0’s in it, you could have (0,0,0) plus a number indicating how many pixels starting the image (if you read line by line from top to bottom) have color (0,0,0).

This leads to a compressed image: an image that contains all, or most, of the information in the original image, but in a more efficient representation. For example, if an original image would have taken 4MB, but the more efficient version takes 400KB, then the compression ratio is 4MB to 400KB, or about 10 to 1.

Complicated compression standards, such as JPEG, use a variety of techniques to compress images. The techniques can be quite sophisticated.

How much can an image be compressed? It depends on a number of factors. For many images, a compression ratio of, say, 10:1 is possible, but this depends on the image and on its use. For example, one factor is how complicated an image is. An uncomplicated image (say, as an extreme example, if every pixel is black<sup>12</sup>), can be compressed a very large amount. Richer, more complicated images can be compressed less. However, even complicated images can usually be compressed at least somewhat.

Another consideration is how faithful the compressed image is to the original. For example, many users will trade some small discrepancies between the original image and the compressed image for a smaller file size, as long as those discrepancies are not easily noticeable. A compression scheme that doesn’t lose any image information is called a *lossless* scheme. One that does is called *lossy*. Lossy compression will give better compression than lossless, but with some loss of fidelity.<sup>13</sup>

---

<sup>12</sup>You might have seen modern art paintings where the entire work is a single color.

<sup>13</sup>See, for example, <http://computer.howstuffworks.com/question289.htm> for examples of the interplay between compression rate and image fidelity.

### 3.10.2 Video

Suppose you have a 10 minute video, 256 x 256 pixels per frame, 24 bits per pixel, and 30 frames of the video per second. You use an encoding that stores all bits for each pixel for each frame in the video. What is the total file size? And suppose you have a 500 kilobit per second download connection; how long will it take to download the file?

This problem highlights some of the challenges of video files. Note the answer to the file size question is  $(256 \times 256)$  pixels  $\times$  24 bits/pixel  $\times$  10 minutes  $\times$  60 seconds/minute  $\times$  30 frames per second = approximately 28 Gb, where Gb means *gigabits*. (This is about  $28/8 = 3.5$  gigabytes.) With a 500 kilobit per second download rate, this will take  $28\text{Gb}/500\text{Kbps}$ , or about 56,000 seconds. This is over 15 hours, longer than many people would like to wait. And the time will only increase if the number of pixels per frame is larger (e.g., in a full screen display) or the video length is longer, or the download speed is slower.

So video file size can be an issue. However, it does not take 15 hours to download a ten minute video; as with image files, there are ways to decrease the file size and transmission time. For example, standards such as MPEG make use not only of image compression techniques to decrease the storage size of a single frame, but also take advantage of the fact that a scene in one frame is usually quite similar to the scene in the next frame. There's a wealth of information online about various compression techniques and standards, storage media, etc. For example, see <http://electronics.howstuffworks.com/question596.htm> and the links there.

### 3.10.3 Audio

It might seem, at first, that audio files shouldn't take anywhere as much space as video. However, if you think about how complicated audio such as music can be, you probably won't be surprised that audio files can also be large.

Sound is essentially vibrations, or collections of sound waves travelling through the air. Humans can hear sound waves that have frequencies of between 20 and 20,000 cycles per second.<sup>14</sup> To avoid certain undesirable artefacts, audio files need to use a sample rate of twice the highest frequency. So, for example, for a CD music is usually sampled 44,100 Hz, or 44,100 times per second.<sup>15</sup> And if you want a stereo effect, you need to sample on two channels. For each sample you want to store the amplitude using enough bits to give a faithful representation. CDs usually use 16 bits per sample. So a minute of music takes  $44,100\text{ samples} \times 16\text{ bits/samples} \times 2\text{ channels} \times 60\text{ second/minute} \times 1\text{ byte}/8\text{ bits} =$  about 10.5MB per minute. This means a 4 minute song will take about 40MB, and an

---

<sup>14</sup>This is just a rough estimate since there is much individual variation as well as other factors that affect this range.

<sup>15</sup>Hz, or *Hertz* is a measurement of frequency. It appears in a variety of places in computer science, computer engineering, and related fields such as electrical engineering. For example, a computer monitor might have a refresh rate of 60Hz, meaning it is redrawn 60 times per second. It is also used in many other fields. As an example, in most modern day concert music, A above middle C is taken to be 440 Hz.

hour of music will take about 630 MB, which is (very) roughly the amount of memory a typical CD will hold.<sup>16</sup>

Note, however, that if you want to download a 40 MB song over a 1Mbps connection, it will take 40MB/1Mbps, which comes to about 320 seconds. This is not a long time, but it would be desirable if it could be shorter. So — not surprisingly — there are compression schemes that reduce this considerably. For example, there is an MPEG audio compression standard that will compress 4 minutes songs to about 4MB, a considerable reduction.<sup>17</sup>

### 3.11 Additional Problems

**Problem 5:** (a) What is the result of the following 8-bit unsigned binary operations? Give the result in unsigned binary. (i) 00101011 + 01100110. (ii) 11101000 – 00110101.  
(b) What is the result of the following 8-bit two’s complement binary operations? Give the result in two’s complement binary. (i) 00110011 + 10011001. (ii) 11110011 + 10010101.  
(c) Convert the following problem into two’s complement binary and perform addition to get the resulting two’s complement binary solution: 80 – 117. Hint: Remember this can be written as an addition problem: 80 + (–117).

**Problem 6:** Suppose you have a color represented as a red, green, blue triple, with each component an integer between 0 and 255 represented as an 8-bit unsigned binary number. The red component is 10010011, the green 11111000, and the blue 00001111. What happens if, in an attempt to make the color lighter, you add 00100000 to each component?

**Problem 7:** Latin alphabetic characters can be represented using their ASCII equivalents. Write the decimal representation of all the characters in “Pei, I.M.” (don’t forget that the punctuation and blank space are considered characters here; but don’t include the quotation marks).

---

<sup>16</sup>See, for example, <http://www.howstuffworks.com/cd.htm> for more information about how CDs work. In general, there is a wealth of web sites about audio files, formats, storage media, etc.

<sup>17</sup>Remember there is also an MPEG video compression standard. MPEG actually has a collection of standards: see, for example, [http://en.wikipedia.org/wiki/Moving\\_Picture\\_Experts\\_Group](http://en.wikipedia.org/wiki/Moving_Picture_Experts_Group).

## 3.12 Problem Solutions

### Introductory Problem:

*Input:* Three integers between 0 and 255, inclusive, that represent respectively the red, green, and blue components of a color.

*Output:* A message stating the largest component or components.

```

1  Get r, g, and b
2  Print 'Largest component'
3  If r > b and r > g
4      Print 'red'
5  Else if g > r and g > b
6      Print 'green'
7  Else if b > r and b > g
8      Print 'blue'
9  Else if r equals g and r > b
10     Print 'red, green'
11 Else if r equals b and r > g
12     Print 'red, blue'
13 Else if g equals b and g > r
14     Print 'green, blue'
15 Else
16     Print 'red, green, blue'
17 Stop

```

### Problem 1:

*Input:* a nonnegative positive integer  $n$ .

*Output:* a list of digits  $b_k, b_{k-1}, \dots, b_1, b_0$  where  $b_0$  is the 1's digit,  $b_1$  the 2's digit, etc., with  $b_k$  the largest (i.e., leftmost digit) in the binary representation of  $n$  (note we aren't adding any 0's to the front to get a predetermined length.)

```

1  Get n
2  Set k to 0
3  While n > 0
4      Set b[k] to the remainder of n/2
5      Set n to the quotient of n/2
6      Set k to k + 1
7  While k > 0
8      Set k to k-1
9      Print b[k]

```

**Problem 2:**  $3987_{10} = 111110010011_2$ ,  $4365_{10} = 1000100001101_2$ , and  $4472_{10} = 1000101111000_2$ .

**Problem 3:** It is possible to get a negative number. For example,  $32_{10} - 64_{10} = -32_{10}$ , which is not in the range 0 to 255.

**Problem 4:**  $FF0581A4_{16}$  has 8 hexadecimal digits, and since each digit takes four bits to store, the number requires 32 bits.

**Problem 5 :** (a)(i)

$$\begin{array}{r} \text{carry :} \quad 1101110 \\ \quad \quad \quad 00101011 \\ \quad \quad + \quad 01100110 \\ \hline \quad \quad \quad 10010001 \end{array}$$

(ii)

$$\begin{array}{r} \text{borrow :} \quad 0110111 \\ \quad \quad \quad 11101000 \\ \quad \quad - \quad 00110101 \\ \hline \quad \quad \quad 10110011 \end{array}$$

(b)(i)

$$\begin{array}{r} \text{carry :} \quad 0110011 \\ \quad \quad \quad 00110011 \\ \quad \quad + \quad 10011001 \\ \hline \quad \quad \quad 11001100 \end{array}$$

(ii)

$$\begin{array}{r} \text{carry :} \quad 1110111 \\ \quad \quad \quad 11110011 \\ \quad \quad + \quad 10010101 \\ \hline \quad \quad \quad 10001000 \end{array}$$

As discussed above, the carry bit that “falls of the left end” can be ignored.

(c) The two’s complement representation for 80 is 01010000. The two’s complement for  $-117$  is 10001011. The resulting calculation is then

$$\begin{array}{r} \text{carry :} \quad 0000000 \\ \quad \quad \quad 01010000 \\ \quad \quad + \quad 10001011 \\ \hline \quad \quad \quad 11011011 \end{array}$$

**Problem 6** :You would get an overflow error on the green component.

**Problem 7** :

Text:           P   e   i   ,   space   I   .   M   .  
ASCII:           80 101 105 44   32   73 46 77 46

## 3.13 Additional Resources

- <http://computer.howstuffworks.com/bytes.htm> . An overview and selected details of data representation and related topics from howstuffworks.



# Chapter 4

## Logic

How do we know if it is true?

### 4.1 Introduction

#### 4.1.1 Introductory Puzzles

Sam Loyd was a famous 19th and early-20th century puzzle author. Here are his instructions for one of his most famous puzzles, “Back from the Klondike,” which (according to Wikipedia) first appeared in the *New York Journal and Advertiser* in April 24, 1898:

Start from that heart [rather than a heart, the grid below uses underlining] in the center and go three steps in a straight line in any one of the eight directions, north, south, east or west, or on the bias, as the ladies say, north-east, northwest, southeast or southwest. When you have gone three steps in a straight line, you will reach a square with a number on it, which indicates the second day’s journey, as many steps as it tells, in a straight line in any of the eight directions. From this new point when reached, march on again according to the number indicated, and continue on, following the requirements of the numbers reached, until you come upon a square with a number which will carry you just one step beyond the border, when you are supposed to be out of the woods and can holler all you want, as you will have solved the puzzle.

```

                4 7 7
              5 4 4 8 3 3 4 6 3
            1 4 5 1 1 1 4 5 1 7 1 3 5
          4 9 4 9 6 7 5 5 5 8 7 6 6 8 5
        3 7 2 9 8 3 5 6 7 3 9 1 8 7 5 8 5
      1 4 7 8 4 2 9 2 7 1 1 8 2 2 7 6 3
    7 2 1 8 5 5 3 1 1 3 1 3 3 4 2 8 6 1 3
  4 2 6 7 2 5 2 4 2 2 5 4 3 2 8 1 7 7 3
4 1 6 5 1 1 1 9 1 4 3 4 4 3 1 9 8 2 7
4 3 5 2 3 2 2 3 2 4 2 5 3 5 1 1 3 5 5 3 7
2 7 1 5 1 1 3 1 5 3 3 2 4 2 3 7 7 5 4 2 7
2 5 2 2 6 1 2 4 4 6 3 4 1 2 1 2 6 5 1 8 8
  4 3 7 5 1 9 3 4 4 5 2 9 4 1 9 5 7 4 8
  4 1 6 7 8 3 4 3 4 1 3 1 2 3 2 3 6 2 4
  7 3 2 6 1 5 3 9 2 3 2 1 5 7 5 8 9 5 4
    1 6 7 3 4 8 1 1 1 2 1 2 2 8 9 4 1
    2 5 4 7 8 7 5 6 1 3 5 7 8 7 2 9 3
      6 5 6 4 6 7 2 5 2 2 6 3 4 7 4
        2 3 1 2 3 3 3 2 1 3 2 1 1
          7 4 4 5 7 3 4 4 7
            3 3 4

```

This set of notes usually contains solutions for the introductory puzzles; however, it won't include one for the Klondike puzzle (you can do an online search for a solution if you wish; better yet, solve it yourself). Instead, we will use this puzzle in another way:

1. Try to solve the puzzle manually. What strategies do you find yourself using?
2. What would an algorithm for solving this puzzle by computer look like? You don't need to write the entire algorithm; just give a high-level description.

Consider the a second, similar puzzle, which does have a solution later in this chapter: You are planning a vacation to Canada with some of your relatives, and you and they vote on which city below to visit. One city got four votes, two got two votes, two got one vote, and the remaining two cities got zero votes. Use logic and the clues below to determine how many votes each city got.

Quebec City  
 Toronto  
 Ottawa  
 Montreal  
 St. John's (Newfoundland)  
 Charlottetown  
 Halifax

Here are the clues. Make sure you explain how you obtained your answer.

- a. Ottawa and Quebec City got different numbers of votes.
- b. Montreal either got the most votes, or it got zero votes.
- c. Quebec City got more votes than Halifax did.
- d. In the list of cities above, each of the two cities that got two votes has a city that got no votes immediately above it in the list.
- e. Either Halifax got one fewer vote than Toronto did, or it got one fewer vote than Ottawa did.

Both of these problems are examples of *logic puzzles*. Such puzzles are useful because (in addition to many people finding them fun) solving them requires the same types of logical reasoning that is needed for many real-world problems in computer science and in other fields.

### 4.1.2 Overview

*Logic* is a field in and of itself. For example, there are entire classes devoted to it. Logic is an essential part of mathematics and philosophy — to name two fields it is closely related to — but also is important elsewhere. And, not surprisingly, it is important in computer science.

“Logic” can mean many different things to computer scientists. For example, computer architects often deal with bit-level logic. A specific example of this is applying logical operations like AND and OR on a bit-by-bit basis to two bytes of data. Computer programmers must deal with program logic such as the different branches a program can take depending on conditions in the program. Computer programmers, system designers, and others use logic to analyze and/or solve problems, analogous to what you might have done on the problems in the last subsection. And theoretical computer scientists must prove results using logically rigorous arguments. The resulting proofs are often quite similar to proofs you might have seen in a math class. Finally, computer scientists, and all people, use logic when communicating with a natural language such as English.

This chapter will explore a variety of different uses of logic in computer science and computer engineering.

### 4.1.3 Why is Logic Important?

Logic might seem like a dry, abstract subject. However, as the examples in the previous subsection show, logic is an essential part of computing. It is important in how computers work, for example, in how computers do low-level operations. It is important in how computer programs work. And it is important in many other ways for computer professionals.

But why is it important to people who want to know about computing, but not at the same level as computer professionals? Here are three reasons:

1. Because logic is so fundamental to how computers work, and how computer scientists and computer professionals think about and solve problems, it is important to have a basic understanding of logic to understand computers and computer science.
2. People use logic in everyday computer use. For example, suppose you want to do an Internet search for information on greyhounds (the dog breed). If you just type in “greyhound” you will also get a number of links related to Greyhound buses. So you can refine your search terms to include “greyhound” but exclude “bus.” This is a (simple) use of logic. As a second example, suppose you maintain a database for a University club. The club asks you to generate a report (based on the database information) of all members who have been in the club for more than two years, who are in the College of Liberal Arts, and who are not currently officers of the club. Specifying this database query involves the use of logic.
3. Logic is important in analyzing and discussing societal issues. Topics such as Internet privacy, Internet forum rules and norms, and computer security often involve intense discussion. How persuasive are arguments people make about such issues? Logic is one (of many) factors people use to gauge the persuasiveness of arguments and the reliability of information.

More generally, the skills you need to solve the various logic problems in this class are skills that computer practitioners often use. Being able to do tasks such as performing logical operations on bitstrings, evaluating a logical expression, and solving logic puzzles are important not only in and of themselves, but also because they are part of, or are related to, many other tasks computer practitioners do.

As this class progresses we’ll continue to use logic as part of certain future topics. For example our next topic, computer organization, once again involves bit-level representation and operations. And later on we’ll use logical techniques as part of solving counting problems, analyzing algorithms, and doing computer programming.

#### **4.1.4 Relation to the Mathematics Liberal Education Requirement**

Logic is very closely related to the mathematics liberal education requirement: logic uses special symbols and notation, operations, rules, etc. Logicians often need to prove results, analyze problems, or do logical calculations. And mathematics relies on logic for the rules of how mathematics “works.”

#### **4.1.5 Relation to the Society and Technology Requirement**

Never before in human history has so much information been so readily available to so many people. And never before has there been so much information that is incorrect, biased, incomplete, or in some other way inaccurate.

With the Internet playing a larger and larger role in how people communicate, for example in discussion of societally important topics, information reliability is critical. When you visit a website, how do you know what you read there is correct, or at least mostly correct? What types of criteria or guidelines do you use? Part of assessing information reliability is based on logic. This logic is not the low-level formal logic that, for example, computer engineers use when designing computer circuits. Instead it is a more informal, higher-level logic. For example, does the information “make sense?” Does the site support any nonobvious claims it makes? Do any claims follow from previous statements or from other information that is known to be true?

### 4.1.6 Goals

Upon completing this topic, you should be able to do the following:

1. Be able to do computations involving logical operators such as AND, OR, and NOT.
2. Be able to evaluate the truth value of given logical statements.
3. Be able to solve simple or moderate-difficulty logic puzzles.
4. Be able to explain and use guidelines for evaluating the reliability of information on web sites.

## 4.2 Logical Operators and Low-Level Logic

### 4.2.1 Introduction to Logical Operators

The last chapter covered how to add binary numbers. But how do computers do basic operations such as binary addition? It isn't something that happens magically. Instead, computers have circuitry, designed by computer engineers, for doing fundamental operations. We'll explore this topic from a computer hardware perspective in the next chapter, which is on machine organization. In this chapter we'll look at one component of that circuitry: *logical operations* (which are implemented as an important part of the circuitry, namely as *logic gates*).

What is a logical operation or logical operator? You are familiar with *arithmetic operators* such as addition, subtraction, exponentiation, etc. You are also familiar with *relational operators* such as greater than, equal to, not equal to, etc. Logical operators indicate logical operations such as AND and OR. In this section we'll look in detail at these operators and operations. In particular, we'll look at a type of logic called *Boolean logic*.

### 4.2.2 Logical AND

Let's look at AND first. Suppose you have two true statements, such as "Mars is a planet in our solar system" and "The Martian atmosphere is roughly 100 times thinner than Earth's." The combined statement "Mars is a planet in our solar system *and* the Martian atmosphere is roughly 100 times thinner than Earth's" is also true. However, suppose you have the statement "The average temperature on Mars is approximately 70° Fahrenheit." This is false (Mars is significantly colder). The combined statement "Mars is a planet in our solar system *and* the average temperature on Mars is approximately 70° Fahrenheit" is false: because of the "and" both parts of the statement must be true for the entire statement to be true.

The AND operation applied to two statements A and B is therefore given by the following table, where each row gives a possible combination of truth values for A and B:

A	B	A AND B
True	True	True
True	False	False
False	True	False
False	False	False

Of course, computers don't have a way of representing 'true' and 'false' directly. They need to encode it in binary in some way. Computer scientists and logicians often use 'T' for true and 'F' for false. So we could, for example, use the ASCII equivalents for 'T' and 'F'. That is one alternative, but it is overkill. We have only two possible values to represent, 'T' and 'F', so we really need only one bit.<sup>1</sup> Therefore we can use a bit value of 1 to represent true, and 0 for false. Here is the table above in terms of 0's and 1's:

A	B	A AND B
1	1	1
1	0	0
0	1	0
0	0	0

Using bit-level logical operations in computers is a powerful idea. Not only are components such as AND gates a building block of computer circuits, i.e., of the actual computer hardware, but logical operations come up surprisingly often in computer programming. For example bit-level logical operations have been used in areas such as computer graphics and image processing. For that reason computer programming languages include logical operations.

As an example of how logical operators are useful in doing other computer operations, consider how computer languages might implement an absolute value function for integers represented using the 8-bit sign/magnitude binary representation discussed in the last

---

<sup>1</sup>Remember one bit allows us to represent two possibilities, two bits allow four, three bits allow eight, etc.

chapter. Recall this representation has the leftmost bit indicating the sign of the number: if the bit is 0, the number is positive; if the bit is 1, the number is negative. Suppose you are working with climate data and are tracking when the first frost occurs at a particular location. Suppose further that the first frost has historically occurred on Oct. 15, and the data value for each year is represented by the number of days the first frost occurred before or after Oct. 15. For example, a data value 2 means the first frost occurred two days after Oct. 15, that is, on Oct. 17. A data value of  $-5$  means the first frost occurred five days before Oct. 15, on Oct. 10.

Suppose your analysis requires not the exact dates the first frosts occurred, but how much those dates varied from the Oct. 15 average. So, for example, Oct. 10 and Oct. 20 are both five days from Oct. 15, so you would want both days represented by  $+5$  rather than having Oct. 10 represented by  $-5$ .

How can a computer change the data to remove any minus signs? One way is to do a simple logical operation with each data item. Specifically, note what happens when we have a value  $X$  and perform the operation  $X \text{ AND } 01111111$ . Since the leftmost bit of  $01111111$  is 0, and a bitwise AND with 0 always yields 0, the sign bit of the result will always be 0. And since a bitwise AND with 1 always yields the bit value you are ANDing with 1, that is  $0 \text{ AND } 1 = 0$  and  $1 \text{ AND } 1 = 1$ , the other seven bits of the result are identical to the corresponding bits in  $X$ .

Here is an example where we have a data value in the top line, and are ANDing it with the  $01111111$  in the second line to get the result in the bottom line. Remember this is a bitwise operation, so it is done column by column and there is no carrying, borrowing, etc. Note the effect is to strip away the leading 1, that is, make the sign bit 0 and therefore make the data value positive.

$$\begin{array}{r}
 1\ 1\ 0\ 0\ 0\ 1\ 1\ 0 \\
 \text{AND } 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\
 \hline
 0\ 1\ 0\ 0\ 0\ 1\ 1\ 0
 \end{array}$$

### 4.2.3 Logical OR and XOR

What are other basic logical operations? There are a number of them. As you might guess another is OR. A statement  $A \text{ OR } B$  is true if  $A$  or  $B$  or both are true, and false if  $A$  and  $B$  are both false. Related to OR is XOR. Sometimes when we use “or” in natural language we mean “one or the other or both”. Sometimes it means “one or the other but *not* both.” For example, which of these two meanings apply to the “or” in the statement “The silent auction will end Friday or Saturday?”<sup>2</sup> With computers it is often important to be very clear about which interpretation we want. So computers distinguish between the regular or operation OR, and the “exclusive or” XOR. Here is a table listing both operations. Note they differ only when  $A$  and  $B$  are both true.

<sup>2</sup>Ambiguities such as those in the statement “I’ll go skiing Friday or Saturday” is one reason language understanding is so difficult for computers (and sometimes for humans as well).

A	B	A OR B	A XOR B
1	1	1	0
1	0	1	1
0	1	1	1
0	0	0	0

Remember that we are giving the operation in terms of bit values 0 and 1. But if you replace 1 with ‘T’ and 0 with ‘F’ you get the operations in terms of true and false.

#### 4.2.4 Logical NOT, Equivalence, and Implication

Another logical operation is NOT. Unsurprisingly, if A is true (equivalently 1), NOT(A) is false (0), and vice versa. NOT takes only a single argument:

A	NOT(A)
1	0
0	1

Two other operators that are useful are the equivalence operator, indicated by a triple horizontal bar  $\equiv$ , and the implication operator, indicated by an arrow  $\rightarrow$ . Equivalence is straightforward: A and B are equivalent if and only if they have the same value:

A	B	A $\equiv$ B
1	1	1
1	0	0
0	1	0
0	0	1

Implication is harder, but is very important since implications occur often. An implication is of the form “If A, then B”. Here is the table:

A	B	A $\rightarrow$ B
1	1	1
1	0	0
0	1	1
0	0	1

Let A be the statement “The Gopher women’s hockey team won their game last night,” and B be the statement “The Gopher women’s hockey team won the league championship.” Then A implies B is “If the Gopher women’s hockey team won their game last night, then they won the league championship.” It’s no surprise that if A is true and B is true (they won the game, and they are the league champions) then the statement A implies B is true. And it is no surprise that if A is true and B is false (the team won but is not the league champion), then A implies B is false.

However, the other two cases can be confusing. Suppose the team lost the game last night, and is not the league champion. So  $A$  is false, and  $B$  is also false. Should the statement “If the Gopher women’s hockey team won their game last night, then they won the league championship” be true or false? Logicians say that statement is logically true.

To understand this, let’s look at another example. Let  $A$  be the statement “You get 98% or more average in this class,” and  $B$  be the statement “You get an A in the class.” Think of the implication “If you get 98% or more average in this class, then you get an A in the class” as a contract. Under which truth values of  $A$  and  $B$  are the terms of the contract upheld (and so the implication statement should be true), and under which truth values are the terms violated (and so the implication statement should be false)?

- If you got a 98% or more average, and you got an A, then the terms are upheld. That is,  $T \rightarrow T$  is  $T$  (true) makes sense.
- If you got a 98% or more average, and you did not get an A, then the terms are violated:  $T \rightarrow F$  is  $F$  (false) makes sense.
- If you got less than 98% average, you might or might not have gotten an A. For example, maybe you got an A because you had a 97% average and that was enough to be in the A range. Or maybe you had a lower grade and got a C+. In either case the contract is not violated because the contract only stipulates what happens if you got a 98% or higher. So  $F \rightarrow T$  does *not* violate the terms of the contract, and so it evaluates to  $T$  (true). Similarly,  $F \rightarrow F$  does not violate the terms of the contract, so it evaluates to  $T$  (true).<sup>3</sup>

In the next section we’ll consider *compound logical statements*: statements consisting of logical substatements joined by logical operators. Before we leave this section, however, here are two additional notes:

First, there are other logical operations. For example, there is a NOR operation. However, the ones above are the most important for this class, and the other operators can be expressed in terms of them. For example, NOR is a combination of NOT and OR. Second, a caution on notation: unfortunately there are many different symbols and names used for logical operations. Here, we will use the symbols above, writing out AND,

---

<sup>3</sup>Here is one more explanation of why logical implication is defined the way it is. Specifically, it shows why if  $A \rightarrow B$  is true, and  $A$  is false, you cannot assume  $B$  must be false. Consider the joke “The philosopher Rene Descartes walks into a bar. The bartender asks him if he’d like a drink. ‘I think not,’ Descartes replies, and then vanishes as if he was never there.”

The joke of course relies on the listener recalling Descartes statement “Cogito, ergo sum” — “I think, therefore I am.” Or we can reword this as “If I think, then I exist.” This is a true implication (leave aside deeper discussions of what it means to think, and what it means to exist). It then relies on humorously equating the the meaning of “I think not” with with the meaning of “I don’t think.” At this stage we have  $A \rightarrow B$  (“If I think, then I exist”) is true, and  $A$  (“I think”) is false. The joke than relies on the logical fallacy that  $B$  (“I exist”) must also be false. The fallacy that if  $A \rightarrow B$  is true, and  $A$  is false, then  $B$  must be false is common enough that logicians have a special fancy name for it: *denying the antecedent*.

OR, XOR, and NOT, and using the symbol  $\rightarrow$  and  $\equiv$  for the implication and equivalence operators, respectively. However, other works use other symbols and other names. As an example AND is sometimes indicated by ‘ $\wedge$ ’, a dot ‘.’, or an ampersand ‘&’, and is sometimes called “logical conjunction” instead of “and.” Wikipedia has a good page on logical symbols at [http://en.wikipedia.org/wiki/List\\_of\\_logic\\_symbols](http://en.wikipedia.org/wiki/List_of_logic_symbols).

### 4.3 Compound Logical Statements

Logical statements can get complicated and can involve more than two (sub)statements and more than one logical operator. For example, suppose you are searching for omelet recipes, and want to include basil or chives (or possibly both), but do not want the recipe to include shallots. If you do an advanced search in Google, you can specify words that the page must contain, a list of words that the page must contain at least one of, or words that should not be on the page. So for example, we could specify the recipe search contains “omelet” and at least one of “basil” or “chive” but does not contain “shallot.” Let us use the following “shorthand”:

A : The page being considered contains the word “omelet”.

B : The page being considered contains the word “basil”.

C : The page being considered contains the word “chive”.

D : The page being considered contains the word “shallot”.

Then the search we are specifying is A AND (B OR C) AND NOT(D).

Suppose a page contains the word “omelet” (A is true), does not contain “basil” (B is false), does not contain “chive” (C is false), and does not contain “shallot” (D is false). Does that page fulfill your search criteria? You can probably see it does not: the logical expression becomes T AND (F OR F) AND NOT(F). Notice F OR F evaluates to F, and NOT(F) evaluates to T. So the statements simplifies to T AND F AND T. Notice this is a statement with three truth values and two AND operations. Absent any parentheses, we evaluate this statement left to right: T AND F evaluates to F, so T AND F AND T simplifies to F AND T, which evaluates to F.

There are rules for which operations to perform first in compound statements. For example, when a compound statement contains operators of different types and no parentheses, then AND operations should be done before OR operations, just as multiplication is done before addition in regular arithmetic. However, because these rules are mildly complicated, we will use parentheses to indicate the order of operations. For example, the rules of logic dictate that in A OR NOT B AND C, first NOT B would be evaluated, then the result of that would be AND’ed with C, and then the OR would be performed. But rather than write A OR NOT B AND C, we will write A OR (NOT(B) AND C). Note this is logically equivalent, but explicitly shows the order of operations.

Suppose, however, we wanted a slightly different statement. The logical expression  $A \text{ OR } (\text{NOT}(B \text{ AND } C))$  is similar to the one in the last paragraph, but the parentheses override the usual order of operations and dictate that we do the AND operation before the NOT. So this is a different logical expression than the one in the previous paragraph. Similarly, due to its parentheses  $(A \text{ OR } (\text{NOT}(B))) \text{ AND } C$  is different still, since it performs the OR before the AND.

**Problem 1:** Does the order of operations matter if all the operations in a compound statement are the same? For example in  $A \text{ AND } B \text{ AND } C$ , does it matter if we evaluate  $A \text{ AND } B$  first versus evaluating  $B \text{ AND } C$ ? How about for implication? Does  $(A \rightarrow B) \rightarrow C$  always evaluate to the same logical value as  $A \rightarrow (B \rightarrow C)$ ?

Here is another example: suppose you have the following statements:

A : The database record being considered is for a person whose last name starts with an ‘F’.

B : The database record being considered is for a person whose first name starts with an ‘F’.

C : The database record being considered is for a person whose occupation is ‘writer’.

D : The database record being considered is for a person who was born before 1900.

Suppose we wanted to return true for each record where the first or last name started with ‘F,’ the person is (or was) a writer, and the person was born in 1900 or later. So, for example, the search criteria should return true for the fiction writer Jasper Fforde (born in 1961), but false for (2013 Nobel Prize winner) Alice Munro.

What would the logical statement be when expressed in terms of A, B, C, D and logical operators? And what would this statement evaluate to under all possible assignments of truth values to A, B, C, and D?

The logical statement would be  $((A \text{ OR } B) \text{ AND } C) \text{ AND } (\text{NOT}(D))$ . To show its value under all possible assignments of truth values, we can construct the *truth table* shown on the next page. This truth table shows all possible combinations of truth values for the A, B, C, and D, as well as the resulting truth value of the entire statement. In this case there are four “logical variables” A, B, C, and D. Each can take on two values, T or F, so there are  $2^4 = 16$  different possible assignments. The table also shows the truth value of intermediate steps.

Truth tables are one way of working with compound logical statements. They have advantages: they provide all possibilities in a structured way. However, they also have disadvantages. For example, sometimes you don’t need to evaluate all possibilities. Moreover, as the table shows, if the logical statement has a large number of different logical variables, then the table will have a large number of rows.

The next problem shows another use for truth tables:

A	B	C	D	NOT(D)	A OR B	(A OR B) AND C	((A OR B) AND C) AND (NOT(D))
T	T	T	T	F	T	T	F
T	T	T	F	T	T	T	T
T	T	F	T	F	T	F	F
T	T	F	F	T	T	F	F
T	T	T	T	F	T	T	F
T	F	T	F	T	T	T	T
T	F	F	T	F	T	F	F
T	F	F	F	T	T	F	F
T	F	T	T	F	T	T	F
F	T	T	F	T	T	T	T
F	T	F	T	F	T	F	F
F	T	F	F	T	T	F	F
F	F	T	T	F	F	F	F
F	F	T	F	T	F	F	F
F	F	F	T	F	F	F	F
F	F	F	F	T	F	F	F

**Problem 2:** Use truth tables to prove or disprove that  $\text{NOT}(A \text{ AND } B)$  always has the same value as  $\text{NOT}(A) \text{ OR } \text{NOT}(B)$ . That is, for all possible assignments of truth values to  $A$  and  $B$  will  $\text{NOT}(A \text{ AND } B)$  always evaluate to the same value as  $\text{NOT}(A) \text{ OR } \text{NOT}(B)$ ?

Often, we are not concerned with all possible truth values, but are given some logical statements whose truth we know or can find out, and are also given a compound logical statement based on the original statements. For example, suppose we have the following:

A : The Art Institute of Chicago has Picasso's painting *The Old Guitarist* in its permanent collection.

B : The Art Institute of Chicago has Grant Wood's painting *American Gothic* in its permanent collection.

C : The Art Gallery of Ontario has a sculpture court of Henry Moore works.

D : Joseph Mallard William Turner's painting *Rain, Steam, and Speed* is in the permanent collection of the Art Gallery of Ontario.

Statements A, B, and C are true, but D is false (the painting is in the National Gallery in London).

Suppose someone claims that "The Art Institute of Chicago has both Picasso's *The Old Guitarist* and Wood's *American Gothic*; or the Art Gallery of Ontario either has

a Henry Moore sculpture court or contains Turner’s *Rain, Steam, and Speed* (but not both).” Write this statement in terms of A, B, C, D and logical operators, and then evaluate its truth value.

To write the statement, note the first part is A AND B, and the second part is C XOR D. (It is a little unclear whether “but not both” applies to the first or second ‘or’; the word ‘either’ is used to indicate it applies to the second.) By the sentence punctuation, we do these operations, and then OR the results. So we get (A AND B) OR (C XOR D). Now since A, B, and C are true (T), and D is false (F), we get

$$\begin{aligned}
 & (A \text{ AND } B) \text{ OR } (C \text{ XOR } D) \\
 = & (T \text{ AND } T) \text{ OR } (T \text{ XOR } F) \\
 = & \qquad \qquad T \text{ OR } T \\
 = & \qquad \qquad T
 \end{aligned}$$

**Problem 3:** Consider the statement “The Art Institute of Chicago has Picasso’s *The Old Guitarist* or Wood’s *American Gothic* (or both); and it is not the case that the Art Gallery of Ontario has both a Henry Moore sculpture garden and Turner’s *Rain, Steam, and Speed*.” Write this statement in terms of A, B, C, D and logical operators, and then evaluate its truth value.

## 4.4 More Bitwise Operations

Let’s extend the material in the last section to bitwise operations. That is, this section explores applying a sequence of bitwise operations to bytes of data.

It will do this by looking at an example from image processing. Suppose you have a *grayscale* image, that is, an image where each pixel is represented a certain gray value rather than an RGB triple. We’ll use a single byte for each pixel with the byte value indicating the pixel’s intensity: for example, 00000000 is black, 10000000 is a mid-level gray, and 11111111 is white. Suppose a pixel’s intensity is given by  $X = 10111101$ . What is the result of the bitwise operation (NOT (X)) AND 11110000?

To answer this question, we first apply NOT to X, changing each 1 to 0 and each 0 to 1, to obtain 01000010. Here is this operation with the NOT of each original bit shown directly below it.

$$\begin{array}{r}
 1\ 0\ 1\ 1\ 1\ 1\ 0\ 1 \\
 \hline
 \text{NOT } 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0
 \end{array}$$

Now we AND this result with 11110000:

$$\begin{array}{r}
 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0 \\
 \text{AND } 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0 \\
 \hline
 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0
 \end{array}$$

**Problem 4:** Give an English explanation of what  $(\text{NOT}(X)) \text{ AND } 11110000$  does to the pixel with original gray value  $X$ .

**Problem 5:** Suppose a pixel's intensity is given by  $X = 10111101$ . What is the result of the bitwise operation  $(\text{NOT}(X)) \text{ XOR } 11100111$ ?

## 4.5 Information Reliability

One of the many ways we judge if information is reliable is whether the information presented is logically convincing. There are different ways that an argument can fail to be convincing. One problem occurs if the facts used as a basis for the argument are incorrect. For example, the claim "Since the Internet was invented in France, countries other than the U.S. should have a larger role in Internet governance" is not very convincing. The reason for this is that the premise "the Internet was invented in France" is not true (French researchers contributed to the creation of the Internet, but the bulk of the work was done in the United States). So even if you believe that countries other than the U.S. should have a larger role in Internet governance you would not find the given argument convincing because the premise is false. Logicians call this an *unsound* argument.

Another way arguments can fail to be convincing is when the conclusion does not follow from the premises. Such an argument is said to be *invalid*. For example, if we are trying to figure out a file error, and we know (i) today the file is not up-to-date; (ii) the file was either updated correctly last night, or an error message was sent to a log file, or a system administrator delayed the update. Then the conclusion "the system administrator delayed the update" does not follow logically since it is possible that an error message was sent to a log file instead.

Here's a tricky point: just because this is an invalid argument does not mean you can reason that the conclusion "the system administrator delayed the update" is false. The administrator might or might not have delayed the update. Based on the argument, however, you don't know which of the two possibilities (the administrator delayed the update or an error message was sent to a log file) occurred. An invalid argument just means that the given argument does not logically support the conclusion being true.

Another problem with applying logic to natural language arguments is that most writing, out of necessity, is not detailed enough that we can apply formal logic. Unless you are writing a rigorous mathematical proof (and probably not even then), you skip steps, assume certain background knowledge, etc.

For example, consider the claim "if a person were to keep walking in a line parallel to the equator, he or she would eventually return to where they started. Note there are some very large assumptions here: the reader knows earth is round(ish), knows what the equator is, understands that 'walk' is meant figuratively rather than literally, realizes that concerns such as the amount of time and resources needed are being excluded from consideration, etc.

When people write, sometimes they err on the side of providing too much detail and

trying too hard to provide conclusions. More common, however, is the opposite error: providing too little detail, or arguments that are too weak. It is too easy for a writer to assume the audience has the same background and mind-set as he or she does.<sup>4</sup> Part of this is the use of logic in writing, and part of it is other considerations.

In conclusion of this section, then, logic plays an important role in information reliability, both in writers convincing their readers, and in readers assessing written information. However, there are also other aspects of information reliability that we will discuss further in other class activities.

## 4.6 Things to Think About

Here are some questions to think about. We will discuss some of these further in class.

1. Think of an area outside of computer science. This could be another field, a hobby, work-related, etc. How is logic used in that area?
2. As one of the examples in this chapter illustrates, logic is often used in database queries. If you have ever used a database program, list some queries you have performed and indicate what logical operators they use.
3. When you write, what techniques do you use to convince the reader that the conclusions you are making are correct?
4. When you read information on the Internet, how do you assess whether that information is reliable?

## 4.7 Additional Problems

**Problem 6:** Let A be the statement “The exercise regimen improved recovery times, on average, by at least one week in knee replacement patients.” Let B be the statement “The improved surgical technique improved recovery times, on average, by at least one week in knee replacement patients.” Fill in the following logic tables:

A	B	NOT A	NOT(A) AND B	(NOT (A) AND B) XOR B
T	T			
T	F			
F	T			
F	F			

---

<sup>4</sup>Some writing advice from David Foster Wallace’s essay “Authority and American Usage”: “(1) Do not presume the reader can read your mind — anything you want the reader to visualize or consider or conclude, you must provide; (2) Do not presume the reader feels the same way that you do about a given experience or issue — your argument cannot just assume as true the very thing you’re trying to argue for.”

A	B	A XOR B	A AND B	(A XOR B) OR (A AND B)
T	T			
T	F			
F	T			
F	F			

**Problem 7:** View the Fortune Magazine 2012 list of the 1000 largest US Corporations, on the web at [http://money.cnn.com/magazines/fortune/fortune500/2012/full\\_list/](http://money.cnn.com/magazines/fortune/fortune500/2012/full_list/). Indicate whether each of the following statements is true or false:

- (i) Hewlett-Packard has more revenue than Apple, and has more profit than International Business Machines (IBM).
- (ii) If Apple has more profit than Microsoft and Amazon.com has more profit than Hewlett-Packard, then Google has more profit than Cisco Systems.
- (iii) Microsoft has more profit than each of the following: Hewlett-Packard, IBM, Dell, Intel, Amazon.com, and Google.
- (iv) Either Cisco Systems has more revenue than Sysco, or Cisco Systems has more profit than Sysco, but not both.
- (v) If Honeywell International has less revenue than Oracle, then either Microsoft has more profit than Oracle or Apple has more profit than Oracle, but not both.
- (vi) Intel has more revenue than Amazon.com or AT&T has more profit than Microsoft, and it is not the case that both Intel and Microsoft each have more profit than Cisco Systems.
- (vii) Intel and Medtronic both appear on the Top 100 list, or AT&T and Verizon Communications each have more profit than Microsoft.

**Problem 8:** Sometime algorithms work at the bit level. (For example, serious encryption techniques use a variety of bit-level operations so the encrypted message is difficult to “crack.”) These algorithms are difficult to trace, but doing so is a good exercise in understanding both algorithms and bit operations. Consider the following algorithm:

*Input:* two 4-bit binary strings  $a$  and  $b$ .

*Output:* one 4-bit binary string  $output$

- 1 Set  $i = 1$
- 2 While  $i < 4$
- 3    $output[i] = b[i + 1]$
- 4   Set  $i = i + 1$
- 5 Set  $output[4] = 1$
- 6 Set  $i = 1$
- 7 While  $i \leq 4$
- 8    $output[i] = output[i] \text{ AND } (\text{NOT}(a[i]))$
- 9   Set  $i = i + 1$
- 10 Set  $tmp = output[4]$

```

11 Set  $output[4] = output[3]$ 
12 Set  $output[3] = tmp$ 
13 Print  $output$ 
14 Stop

```

Note: In the pseudocode for this problem we use the notation  $output[i]$  to mean the value of the  $i$ th bit of binary string  $output$ , read left to right. So  $output[1]$  is the most significant (leftmost) bit and  $output[4]$  is the least significant (rightmost) bit, as shown in this figure:

most significant	$\implies$	least significant				
<table border="1" style="border-collapse: collapse; margin: 0 auto;"> <tr> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">2</td> <td style="padding: 2px 5px;">3</td> <td style="padding: 2px 5px;">4</td> </tr> </table>			1	2	3	4
1	2	3	4			
smallest index	$\implies$	largest index				

For example, if  $a = 0011$ , then  $a[1] = 0$  and  $a[4] = 1$ . If  $b = 1010$ , then  $b[1] = 1$  and  $b[4] = 0$ .

(i) Trace through the algorithm for input  $a = 0011$ ,  $b = 1010$ . Specifically, (a) show the values of  $i$  and  $output$  immediately before each time Line 4 is executed, (b) show the values of  $i$  and  $output$  immediately before each time Line 9 is executed, and (c) show what is printed. In part (a), if a variable is not yet assigned a value when Line 4 is executed, leave the value for that variable blank.

(ii) Trace through the algorithm for input  $a = 1001$ ,  $b = 1111$ . Specifically, (a) show the values of  $i$  and  $output$  immediately before each time Line 4 is executed, (b) show the values of  $i$  and  $output$  immediately before each time Line 9 is executed, and (c) show what is printed. In part (a), if a variable is not yet assigned a value when Line 4 is executed, leave the value for that variable blank.

**Problem 9:** Suppose you can evaluate the logical expression  $A \text{ XOR NOT } B \text{ AND } A$  in any order you want. How many different possible orders of evaluation are there?

## 4.8 Problem Solutions

### Second Introductory Problem:

Here are the vote totals, followed by a line of reasoning to obtain the totals. Other lines of reasoning were possible as well.

Quebec City	4
Toronto	0
Ottawa	2
Montreal	0
St. John's	2
Charlottetown	1
Halifax	1

*Observation 1:* From Clue (b) Montreal either had 0 or 4 votes.

*Observation 2:* From Clue (c) Quebec City had more votes than Halifax, meaning Quebec City did not get 0 votes and Halifax did not get 4 votes.

*Observation 3:* From Clue (d) we get the following. Note that since there are two cities with two votes, and two with no votes, every city with two votes must be immediately below a city with no votes, and every city with no votes must be immediately above a city with 2 votes. (i) Quebec City did not get 2 votes since it has no city above it in the list to get 0 votes. (ii) Toronto did not get 2 votes, since it is below Quebec City, which by Observation 2 did not get 0 votes. (iii) Ottawa did not get 0 votes since it is above Montreal, which by Observation 1 did not get 2 votes. (iv) Halifax did not get 0 votes since there is no city below it to get 2 votes.

*Observation 4:* Halifax got 1 vote. This is because it did not get 0 or 4 votes by observations 3(iv) and 2, respectively. Moreover, by Clue (e) there was a city with one more vote than Halifax. Since no city got 3 votes, Halifax could not have gotten 2 votes. The only option left is it got 1 vote.

*Observation 5:* Ottawa must have gotten 2 votes. This is because by Clue (e) either Toronto or Ottawa got one more vote than Halifax. By Observation 4 Halifax got 1 vote, so either Toronto or Ottawa got 2. However, Toronto did not get 2 votes by Observation 3(ii). So Ottawa got 2 votes.

*Observation 6:* By Observation 5 and Clue (d), Toronto got 0 votes.

*Observation 7:* Quebec City got 4 votes. This is because, by Clue (c), it must have more than the number of votes Halifax did, which is 1 by Observation 4. Moreover, Quebec City did not get 2 votes by Observation 3(i). So the only option left is 4 votes.

*Observation 8:* Montreal got 0 votes. This is because it had either 0 or 4 votes. But only one city got 4 votes, and by Observation 7 that city was Quebec City. So the only option left for Montreal is 0 votes.

*Observation 9:* St. John's got 2 votes. This is because it is immediately below a city, Montreal, which by Observation 8 got 0 votes. So by Clue (d) and the comment in Observation 3, St. John's must have gotten 2 votes.

*Observation 10:* Only one city, Charlottestown, remains. And all the vote numbers are accounted for except for one city with 1 vote. So Charlottestown must have gotten 1 vote.

### **Problem 1:**

The order is sometimes, but not always, important. AND is independent of the order used:  $A \text{ AND } B \text{ AND } C$  will evaluate to true, regardless of which AND is evaluated first, if and only if all of A, B, and C are true. Similarly,  $A \text{ OR } B \text{ OR } C$  will evaluate to true, again regardless of the order, if and only if at least one of A, B, or C is true. However, this order independence is *not* true of all operations:  $(A \rightarrow B) \rightarrow C$  does not always evaluate to the same truth value as  $A \rightarrow (B \rightarrow C)$ . For example, let A be false, B be false, and C be false. Then  $(A \rightarrow B) \rightarrow C$  simplifies to  $(F \rightarrow F) \rightarrow F$ , and so to  $T \rightarrow F$ , and so to F. But  $A \rightarrow (B \rightarrow C)$  simplifies to  $F \rightarrow (F \rightarrow F)$ , and so to  $F \rightarrow T$ , and so to T.

**Problem 2:**

Here is a truth table showing the equivalence. Note the two bold columns have identical values:

A	B	A AND B	<b>NOT (A AND B)</b>	NOT(A)	NOT(B)	<b>NOT(A) OR NOT(B)</b>
T	T	T	<b>F</b>	F	F	<b>F</b>
T	F	F	<b>T</b>	F	T	<b>T</b>
F	T	F	<b>T</b>	T	F	<b>T</b>
F	F	F	<b>T</b>	T	T	<b>T</b>

**Problem 3:**

The statement is (A OR B) AND (NOT (C OR D)). Notice that statement uses OR rather than XOR, and that the punctuation and phrasing indicates the order of operations. Substituting the truth values gives

$$\begin{aligned}
 & (A \text{ OR } B) \text{ AND } (\text{NOT } (C \text{ AND } D)) \\
 = & (T \text{ OR } T) \text{ AND } (\text{NOT } (T \text{ AND } F)) \\
 = & \quad \quad T \text{ AND } (\text{NOT } (F)) \\
 = & \quad \quad \quad T \text{ AND } T \\
 = & \quad \quad \quad \quad T
 \end{aligned}$$

**Problem 4 :**

The NOT inverts the intensity, so intense gray values would go to dark ones, and vice versa (for example white goes to black, and black to white). The AND then takes the result and “rounds down” to the nearest multiple of 16 (for example, a value of  $18_{10}$  is rounded to  $16_{10}$ , a value of  $89_{10}$  to  $80_{10}$ , etc.).

**Problem 5:**

Applying (NOT (X)) XOR  $11100111$  to  $X = 10111101$  gives  $01000010$  for applying NOT to X, and then the XOR gives

$$\begin{array}{r}
 01000010 \\
 \text{XOR } 11100111 \\
 \hline
 10100101
 \end{array}$$

**Problem 6:**

A	B	NOT A	NOT(A) AND B	(NOT(A) AND B) XOR B
T	T	F	F	T
T	F	F	F	F
F	T	T	T	F
F	F	T	F	F

A	B	A XOR B	A AND B	(A XOR B) OR (A AND B)
T	T	F	T	T
T	F	T	F	T
F	T	T	F	T
F	F	F	F	F

**Problem 7:**

First, here are the relevant rows from the Fortune 500 table.

Rank	Company	Millions of dollars	
		Revenue	Profit
10	Hewlett-Packard	127,245.0	7,074.0
11	AT&T	126,723.0	3,944.0
15	Verizon Communications	110,875.0	2,404.0
17	Apple	108,249.0	25,922.0
19	International Business Machines	106,916.0	15,855.0
37	Microsoft	69,943.0	23,150.0
44	Dell	62,071.0	3,492.0
51	Intel	53,999.0	12,942.0
56	Amazon.com	48,077.0	631.0
64	Cisco Systems	43,218.0	6,490.0
69	Sysco	39,323.5	1,152.0
73	Google	37,905.0	9,737.0
77	Honeywell International	37,059.0	2,067.0
82	Oracle	35,622.0	8,547.0
164	Medtronic	15,933.0	3,096.0

For each part, to decide whether a statement is true we

- split the statement into logically simple parts and give each part a symbolic name such as  $a$  or  $b$ ;
  - evaluate the truth of each simple part;
  - express the complete statement in terms of logical operations (AND, OR, NOT, etc.) and simple parts;
  - evaluate the logical expression.
- (i) a: Hewlett-Packard has more revenue than Apple: **true**  
 b: Hewlett-Packard has more profit than International Business Machines: **false**

a AND b:  
 true AND false  
 ANSWER: false

- (ii) a: Apple has more profit than Microsoft: **true**  
 b: Amazon.com has more profit than Hewlett-Packard: **false**  
 c: Google has more profit than Cisco Systems: **true**

IF (a AND b), THEN c:  
 IF (true AND false), THEN true  
 IF false, THEN true  
 ANSWER: true

- (iii) a: Microsoft has more profit than Hewlett-Packard: **true**  
 b: Microsoft has more profit than IBM: **true**  
 c: Microsoft has more profit than Dell: **true**  
 d: Microsoft has more profit than Intel: **true**  
 e: Microsoft has more profit than Amazon.com: **true**  
 f: Microsoft has more profit than Google: **true**

a AND b AND c AND d AND e AND f  
 true and true and true and true and true and true  
 ANSWER: true

- (iv) a: Cisco Systems has more revenue than Sysco: **true**  
 b: Cisco Systems has more profit than Sysco: **true**

a XOR b  
 true XOR true  
 ANSWER: false

- (v) a: Honeywell International has less revenue than Oracle: **false**  
 b: Microsoft has more profit than Oracle: **true**  
 c: Apple has more profit than Oracle: **true**

IF a THEN (b XOR c)  
 IF false THEN (true XOR true)  
 IF false THEN false  
 ANSWER: true

- (vi) a: Intel has more revenue than Amazon.com: **true**  
 b: AT&T has more profit than Microsoft: **false**  
 c: Intel has more profit than Cisco Systems: **true**  
 d: Microsoft has more profit than Cisco Systems: **true**

(a OR b) AND NOT(c AND d)  
 (true OR false) AND NOT(true AND true)  
 true AND NOT(true)  
 true AND false  
 ANSWER: false

- (vii) a: Intel appears on the Top 100 list: **true**  
 b: Medtronic appears on the top 100 list: **false**  
 c: AT&T has more profit than Microsoft: **false**  
 d: Verizon Communications has more profit than Microsoft: **false**

(a AND b) OR (c AND d)  
 (true AND false) OR (false AND false)  
 false OR false  
 ANSWER: false

### Problem 8.

- (i) Here is the trace at Line 4, at line 9, and the final output:

Line 4	i	output[1]	output[2]	output[3]	output[4]
	1	0			
	2	0	1		
	3	0	1	0	
Line 9	1	0	1	0	1
	2	0	1	0	1
	3	0	1	0	1
	4	0	1	0	0
Printed		0	1	0	0

- (ii) Here is the trace at Line 4, at line 9, and the final output:

Line 4	i	output[1]	output[2]	output[3]	output[4]
	1	1			
	2	1	1		
	3	1	1	1	
Line 9	1	0	1	1	1
	2	0	1	1	1
	3	0	1	1	1
	4	0	1	1	0
Printed		0	1	0	1

**Problem 9:** There are three possibilities:  $(A \text{ XOR } \text{NOT}(B)) \text{ AND } A$ ;  $A \text{ XOR } (\text{NOT}(B) \text{ AND } A)$ ;  $(A \text{ XOR } \text{NOT}(B \text{ AND } A))$ .

## 4.9 Additional Resources

Here are some additional resources:

- <http://computer.howstuffworks.com/boolean1.htm> . This file from howstuffworks explains in detail how the Boolean logic in this chapter is important in computer hardware. This not only explains this connection further, but also explains some items that will appear briefly in the next chapter.
- [http://www.nlm.nih.gov/bsd/disted/pubmedtutorial/020\\_350.html](http://www.nlm.nih.gov/bsd/disted/pubmedtutorial/020_350.html) . A fairly simple tutorial from the National Institute of Health's U.S. National Library of Medicine explaining simple searches of medical databases. The examples show use of AND, OR, and NOT.
- [http://en.wikipedia.org/wiki/Reliability\\_of\\_Wikipedia](http://en.wikipedia.org/wiki/Reliability_of_Wikipedia) . This is a Wikipedia article about Wikipedia. It is fairly long, but not all sections are relevant for this class. In addition to this page, Wikipedia also contains other pages about its reliability, such as [http://en.wikipedia.org/wiki/Wikipedia%3AAcademic\\_use](http://en.wikipedia.org/wiki/Wikipedia%3AAcademic_use).



# Chapter 5

## Machine Organization

It is still all 0's and 1's.

### 5.1 Introduction

#### 5.1.1 Introductory Problem

Suppose you work at an arboretum. You have a list of 100 data records for trees. Each record contains three data: the first is the type of tree, stored as a 6-character string (assume you have abbreviated codes for the tree names so you do not have to store the entire, potentially lengthy, name). The second is the year the tree was planted, stored as a 16-bit integer. The third is the diameter of the tree the last time it was measured. This is stored as a 32-bit floating point number.

- (a) How much memory does the list take?
- (b) Assume all the records are stored one right after the other, with the first record at memory location (given in hexadecimal, since memory locations are often given in hexadecimal) `3b2201aa`. What memory location or locations are occupied by the last record? Give your answer in hexadecimal.

#### 5.1.2 Overview

How do computers work?

In the past few chapters you've seen parts of this puzzle. The chapter on data representation used binary numbers for storing different types of data. The chapter on logic briefly mentioned implementing computer operations in circuits containing logic gates. And the chapter on algorithms used sequences of (higher-level) instructions to solve problems and perform tasks. But how do the algorithms' instructions work on a lower level? And how do all these parts — data, low-level operations, and high-level instructions — come together?

Algorithms are implemented in a computer language such as Java, C++, or Python. So one answer to the first question in the last paragraph is that algorithms are entered into the computer as text representing the computer program. And in the data representation chapter we saw that computers can represent text in ASCII. But this raises still other questions. For example, how does a computer distinguish between text representing computer instructions, and text representing data? And even if an instruction is recognized as an operation and not as data, how does a computer, on a low level, actually perform that operation?

Computer instructions, on a low level, are still all 0's and 1's. While instructions in an algorithm specification are implemented in a computer language such as Java, these instructions are eventually converted into *machine instructions*, which have a binary equivalent. For example, for a particular computer a bitstring 0001101011 might mean “check if the value of the variable I am about to give you (the location of) is equal to 0.”

Note this means, on one hand, that computer instructions are another type of data, stored in computer memory along with the other data the computer holds. On the other hand, computer instructions are a special type of data, and do not use ASCII equivalents; instead they use other binary equivalents called *operation codes*, or *opcodes* for short. And computers must work with these operation codes differently than they do “regular” data.

The purpose of this chapter is to delve further into the inner workings of computers. Although this will be a brief and simplified version of how computers work,<sup>1</sup> it will nonetheless cover key ideas about computers, computer science, and computer engineering. In summary, in this chapter we'll look questions such as

- How, on a low level, do computers work?
- How do computers represent instructions?
- How are computer hardware and software related?
- What are the important parts of a computer, and what is important to know about them?

### 5.1.3 Motivation

How do things work? We usually do not need to understand the “inner workings” of technological devices in order to use them: how they work is hidden away behind a simple interface. Think, for example, of driving a car. You need to know how to start the car, how to steer it, how to stop it, etc. You do not need to know how the engine works, how the electrical system works, or the exact composition of the tires. That all is hidden.

---

<sup>1</sup>How computers work is very complicated. For example, there are entire classes devoted to subparts of this topic. Here are a few at the University of Minnesota: CSci 2021, Machine Organization; EE 2001, Introduction to Circuits and Electronics; EE 2301, Introduction to Digital System Design; EE 2361, Introduction to Microcontrollers; CSci 4203/EE 4363, Computer Architecture.

Similarly, to be a computer user of office software such as spreadsheets, of a web browser, or of a statistical analysis package, etc. you do not need to know how the computer works “under the hood.”

On the other hand, there are times when it is useful to know about computer hardware and the inner workings of a computer. Suppose you are buying a new computer and it is advertised as having “cache memory.” What does that mean? Is it good or bad? Or suppose it has a one terabyte hard disk. Is that a large amount of memory? Or suppose a program tells you there is a “bus error.” What do busses have to do with computers? Knowing at least a little about computer hardware can be helpful when buying computers, or when working with some programs.

Moreover, this is a chapter where a number of key ideas in computer science come together. Specifically, this chapter, more than any other chapter, explains how computers pull together algorithms, data, and logic, and actually do things.

#### **5.1.4 How This Topic Relates to the Mathematics Requirement**

In this chapter you will again see the connections between computing and mathematics that appeared in previous chapters. These include binary numbers, carefully stepping through a computational process, and doing arithmetic problems related to memory space or access time.

#### **5.1.5 How This Topic Relates to the Society and Technology Theme**

The increasing effects of computers on society are based on computers’ increasing capabilities. Twenty years ago laptops, streaming audio and video, social networks site, etc. were uncommon, if they existed at all. However, continual and dramatic improvements in computer capabilities, for example increases in computer speed and in computer memory, have made possible what we take for granted nowadays.

To understand how computers have advanced — and what further advances are likely in the future — we need to know the basics of how computers work. So, for example, we need to know what the main parts of a computer are, we need to know something about computer speed, we need to know about computer memory, etc. This chapter discusses those basics. And in the next chapter, which is on Moore’s Law, we will focus specifically on how computer hardware has advanced and is advancing.

#### **5.1.6 Goals**

Upon completing this topic, you should be able to do the following:

1. Be able to list different parts of a computer (this includes explaining any relevant terms), why they are important, and how they are related.

2. Be able to explain, on a low level, how computers work.
3. Be able to relate computer component characteristics to common tasks. (Example: is 256K of memory enough to store the text from a normal sized book?).
4. Be able to solve arithmetic problems involving computer components. (Example: Suppose you download a 3MB file over a 250Kbps connection. How long will it take?).
5. Be able to read, translate, and trace through the execution of a given short sequence of machine instructions.

## 5.2 An Analogy

Understanding the terminology, motivation for, and important characteristics of computer parts can be difficult. So we will start with an analogy.

Suppose you do woodworking, and have a work area including a workbench, a set of tools and supplies adjacent to your workbench, and lesser-used supplies and tools located elsewhere in the room. Suppose further you are creating a holiday decoration (pick your favorite holiday). To construct the decoration you are following a set of instructions. You put a printed copy of the instructions on the workbench so you can easily follow it. And you put the wood and other supplies you will be using on the side of the workbench.

(If woodworking doesn't appeal to you, then think about any process using materials, tools, and instructions; possibilities include creating a garden, constructing a floral arrangement, or restringing a guitar.)

The instructions are step-by-step, so you follow them one at a time. Sometimes a step involves taking material and performing an operation on it (e.g., "cut a 1/4 inch diameter dowel to a length of 4 inches"). Note this involves finding the specific piece, getting the tool (a saw, presumably) from the nearby set of tools, cutting the piece, returning the saw to its place (or you might set it down directly on the workbench if you will be using it again very soon), and setting the cut piece aside (unless it is being used again very soon). Sometimes the step involves a number of substeps (consider, for example, all the substeps involved in drilling a hole in a piece of wood: marking the location of the hole, finding the right size drill bit, etc.). Usually a step requires getting supplies or tools that are at hand; but sometimes it involves supplies or tools farther away. For example, suppose a step asks you to use a clamp that you use so infrequently that you store it in a drawer on the other side of the room. Then you need to move to the other side of the room, locate the clamp, and carry it back to the workbench. Sometimes you might even need to go farther away (e.g., to a hardware store) to get additional material or tools.

How is this like a computer running a program? Like all analogies, this one isn't an exact parallel, but it does serve to illustrate several key points:

- There is a center of activity (the workbench) where you do the work. In a computer there is a *CPU* (*central processing unit*) where most of the operations are done.

- The overall operation is broken into a sequence of steps. Most steps involve reading the current instruction, understanding what it means, retrieving any needed supplies and tools, performing the operations, placing the result somewhere, returning any tools used to their designated locations, and then going to the next instruction. This process is repeated again and again until all the needed steps have been performed. Similarly a computer loads a program into memory, and follows the program's instructions one by one. Individual steps involve decoding the instruction, fetching any needed values from wherever they are stored, performing the specified operation, and then placing the results somewhere they can be accessed later.
- There are different locations where items are kept. Sometimes supplies or tools are set near the center of the workbench when they are part of the current or an upcoming step. Sometimes they are kept on the side of the workbench for easy access. Some are in the adjacent storage area. Some are elsewhere. Note the closer an item is, the less time it takes to find and fetch it. Ideally, all items you need are located nearby and are easy to find. However, many projects will involve getting at least some items from further away.

Similarly, computers contain different types of memory such as register memory, cache memory, main memory, and secondary memory (these types will be explained later in this chapter). Some memory can be accessed very quickly, but is smaller and more expensive in cost. Other memory is cheaper and more plentiful, but has a significantly slower access time.

- There are a variety of different types of supplies and tools. Similarly, a computer program involves data (including different types of data), program instructions from the program the computer is running, and instructions from other helper programs that are needed for the program to run.
- You are “managing the process” by gathering all the needed items, putting them in the appropriate locations, finding and fetching items as needed, stepping through the instructions one-by-one and remembering where you are in the instruction list, returning items to their locations when you are done with them, etc.

Similarly in a computer there is a “manager” — the *operating system* — that keeps track of everything and performs those higher-level operations such as moving data from memory to the CPU.

The remainder of this chapter will explore some of these computer parts and processes in more detail.

## 5.3 Chapter Structure

This chapter will have a slightly different structure than most other chapters in this set of notes. So this section provides a “roadmap” to the remainder of this chapter and

instructions about some online resources.

Machine organization is an area where there are a number of useful introductory-level online resources. So we will rely heavily on those resources, asking you to read them for some basic information. This chapter will provide some additional basic material, and provide some example problems.

The three web resources listed below provide much of the basic material you will need to know. Each of these makes some important points about computers, and two of them also contain some useful diagrams. However, the sites also contain more information than we need: make sure you understand the key points, but don't worry if you don't understand all the details. In particular, focus on the following three items from the sites:

- An understanding of the key parts of computers. The sites mention a number of components, so you should understand both the names and purposes.
- An understanding of how the parts relate to each other. Two of the sites have very useful diagrams illustrating key relations.
- Tradeoffs. Two sites mention tradeoffs such as memory cost versus access time or processor speeds versus computer cost.

The “Terminology” section (5.4) below contains a brief explanation of important terms; this might be useful as you read the sites. The sites also contain links to additional sites with even further information. You don't need to follow these links, although you are certainly welcome to explore them if you like.

Here are the three sites you should read:

- [http://en.wikibooks.org/wiki/Computers\\_for\\_Beginners/Buying\\_A.Computer](http://en.wikibooks.org/wiki/Computers_for_Beginners/Buying_A.Computer). This is a Wikibooks page on buying a computer. While some material here needs updating (a common problem when discussing computer capabilities since they change quickly) and sometimes the page uses too much terminology for our purposes, this resource is nonetheless a good introduction to the different parts of a computer and the range of differences that exist in those parts. Read the entire page.
- <http://computer.howstuffworks.com/computer-memory1.htm>. This is the *HowStuffWorks* “How Computer Memory Works” article. Pay particular attention to the different types of memory (e.g., main memory, register, cache, hard disk); how they work in general; and the motivation for, and characteristics of each. Read this entire article.
- <http://computer.howstuffworks.com/microprocessor.htm>. This is the *HowStuffWorks* “How Microprocessors work” article. Read the first five pages, paying particular attention to the diagram and description, on page 3, of the different parts of the central processing unit. The page 5 description of microprocessor instructions is also useful. A section below will provide a further explanation of such instructions on an even lower level.

The remainder of this chapter then provides the following additional material:

- A list of machine organization terms. This will provide a quick reference for terms that come up in the readings and elsewhere in the class.
- An explanation of memory addresses and storing data in computer memory.
- Information on machine instructions.
- Some quick comments and tables on memory sizes and processor speeds.
- Brief comments on system software.
- Brief comments on computational models.
- Some further questions to think about.
- Example problems and their solutions.

## 5.4 Machine Organization Terms

Because of all the terminology and concepts associated with machine organization, this section lists a number of terms and give a brief explanation of each. While not all terms are included here, the most important are.

- *ALU* (arithmetic/logic unit): part of a processor that handles low-level arithmetic (e.g., addition) or logic (e.g., comparing two numbers).
- *address field*: bits used to designate the memory location associated with an instruction. For example, the command LOAD A means load the contents of memory address A into the register; and in machine code the location of ‘A’ must be represented in binary in the address field for that command.
- *address field width*: the number of bits in the address field. This number must be enough that each memory location in the computer’s main memory has its own unique address. An  $n$ -bit memory field therefore allows the computer to have at most  $2^n$  memory locations.
- *assembly language*: a low-level language that has a symbolic name such as LOAD or COMPARE for each machine instruction.
- *bus*: a means for moving data around within a computer.
- *cache memory*: special (but limited size) memory that allows faster access than main memory.

- *compiler*: one type of program that translates program code written in a high-level language into machine code.
- *CPU* (central processing unit): the main processor in the computer.
- *FLOPS* (floating point operations per second): one measure of a computer's speed. Powerful computers' speeds are in the giga-FLOPS or even tera-FLOPS range.
- *GPU*: (graphical processing unit): a processor that handles the computationally-intensive graphics operations such as drawing graphics objects to the screen.
- *Hertz*: a measure of frequency. MHz, or megahertz, is millions of cycles per second. GHz, or gigahertz, is billions of cycles per second. Processor speeds are usually in the MHz or GHz range.
- *high-level language* (or HLL): a language such as Java, Python or C++. One characteristic of such languages is that they contain keywords such as *for*, *print*, etc. that are more easily understood by humans than opcodes are.
- *instruction set*: the set of low-level commands for a processor. These command include commands such as LOAD or JUMP.
- *interpreter*: one type of program that translates program code written in a high-level language into machine code. For example, you will use a Python interpreter in this class.
- *machine code*: program code which has been compiled or interpreted into its binary representation. The machine code version of a program is also called the executable version or the binary version.
- *main memory* (or primary memory): the “working memory” of a computer. For most personal computers main memory measures in hundreds of megabytes or in gigabytes, and holds programs as they run, data that is being used, etc.
- *memory*: in general, any part of the computer used to store values, including secondary memory, main memory, cache memory, or registers. However, often the term “memory” is used to refer only to main memory.
- *memory address*: the numeric identifier (usually given in binary or hexadecimal, but sometimes in decimal) of the location of a byte in memory.
- *memory value*: the value of the bits stored in a given memory location. Each byte in memory has both an address and a value.
- *MIPS*: millions of instructions per second. One measure of a computer's speed.

- *opcode*: (or *op code*): a binary string that represents a command from a processor's instruction set.
- *opcode width*: the number of bits used for opcodes. This will vary from machine to machine. The number of bits used must allow each command in the instruction set to have its own unique opcode; so an  $n$ -bit opcode width allows an instruction set of at most  $2^n$  commands.
- *program counter*: when the computer is executing a program, the program counter holds the memory address of the instruction currently being executed. Once that instruction is finished, then the program counter is updated to the address location of the next instruction.
- *register*: special, very small-sized memory in the CPU. Registers are used to temporarily hold values during low-level computer operations.
- *storage*: a term that usually refers to secondary memory (hard drives, CDs, floppy disks (for older computers), magnetic tapes, thumb drives, etc.).

## 5.5 Memory Addresses

To understand how computers work, it is important to understand how computers store data. Specifically, in this section we'll look at how computers store variables. Much of this applies to more general data as well.

Suppose we are writing a computer program that analyzes poetry. One function we want the program to do is find the average number of lines of a group of poems. If we were to do this, the program would might have the following outline:<sup>2</sup>

```

Set numberOfPoems to 0
Set lineSum to 0
While there are still poems to consider
    Get the next poem
    Set numberOfPoems to numberOfPoems + 1
    Set n to be the number of lines in that poem
    Set lineSum to lineSum + n
If numberOfPoems equals 0
    Print 'No poems to analyze.'
Else
    Print 'Average Number of Lines: ', lineSum/numberOfPoems

```

---

<sup>2</sup>Note this is an outline, but to be a valid pseudocode specification some lines would need to be refined further.

Remember from Chapter 1 that `numberOfPoems`, `lineSum`, and `n` are called *variables* since their value might vary as the algorithm progresses. This is in contrast to *constants*, whose values remain the same throughout a program. For example, if you were writing a program to compute the volume of a sphere, you would use a formula involving the number  $\pi$ , which is approximately 3.14159. The number  $\pi$  is a constant: its value does not change.<sup>3</sup>

To understand how computers work with variables, let's assume further that some of the poems you are analyzing have disputed lines, i.e., lines which might be in some publications of the poems, but which experts are unsure actually belong to the poem. Suppose you have a count `numDisputedLines` and do the following:

```
print 'Do you wish to add the disputed lines (y/n)?'
get ch
if ch equals 'y'
    lineSumAug = lineSum + numDisputedLines
else
    lineSumAug = lineSum
```

Suppose at the start of this code `lineSum` had value 1066, and `numDisputedLines` had value 42. Suppose the user didn't want to add the disputed lines, so the `else` part is executed. So now we have both `lineSum` and `lineSumAug` equal to 1066. Suppose further that, for whatever reason, the value of `lineSum` is then changed by the line

```
lineSum = lineSum + 10
```

This changes the value of `lineSum` to 1076. Now what is the value of `lineSumAug`? Should it be 1066, because that is what it is set to in the `else` part above? Or should it be 1076: does equating the two variables in the `else` part link them so `lineSumAug` changes whenever `lineSum` does?

The answer here is that `lineSumAug` still equals 1066. While this might seem like a silly example, it hinges on an important point, namely how the computer interprets the line `lineSumAug = lineSum`. It is setting the *value* of `lineSumAug` to the value of `lineSum`; but it is not linking the *names* of the two variables.

Variables in computer programs, therefore, are not just values, but involve both names and values. Moreover, they are even more than names and values. For example, in many languages it is possible to have two different variables, in different parts of the program, that share the same name. Therefore, variables are a collection of information, including the variable's value, name, location in memory when the program is running, and type.

Let's look at these further. You can think of computer memory<sup>4</sup> as a long list of locations. Each location is a byte (equivalently, 8 bits), and computers will put values

---

<sup>3</sup>As an aside, we usually approximate  $\pi$  to just a few digits, but  $\pi$  is an irrational number — it goes on and on forever without any consecutively repeating pattern. People have used computers to calculate trillions of digits of  $\pi$ . For more information, see the many informative web sites such as <http://en.wikipedia.org/wiki/Pi> or <http://www.joyofpi.com/pilinks.html>.

<sup>4</sup>In this section “memory” refers to main memory.

into, modify values within, or take values out of those locations. When a computer program runs, it needs memory for doing what it needs to do, and this includes needing space for its variables. Let's suppose we're working with the number of lines example above. The computer system would therefore assign memory space for `lineSum` and `lineSumAug`, along with the other variables involved. How do we refer to or identify this space?

Each location in memory has an *address*. In some ways this is similar to house addresses, since each house on a street often has an identifying number. But this analogy only goes so far. For example, you will probably not be surprised that computer scientists and computer engineers will often write memory addresses in binary or (because the number of bits is usually large) hexadecimal.

Let's suppose we have a small device that has 65,536 memory locations. Note this is a power of 2, namely  $2^{16}$ ; this is not a coincidence since maximum memory sizes for computing devices are almost always powers of 2. The device therefore needs an *address field width* of 16; that is, each address consists of 16 bits as shown in the left-hand column below. (If we wished, we could use hexadecimal or even decimal to represent the addresses.) All the values in the right column are set to 00000000. In an actual computer, these values would usually be different.

Address	Value
0000000000000000	00000000
0000000000000001	00000000
0000000000000010	00000000
0000000000000011	00000000
...	...
1111111111111100	00000000
1111111111111101	00000000
1111111111111110	00000000
1111111111111111	00000000

The computer does not actually store the addresses in the left hand side; it stores only the values and can figure out the addresses as needed. This is emphasized in the table above by the missing left-side boundary line.<sup>5</sup> However, we'll usually include addresses in memory diagrams since we'll need to use the addresses to refer to specific memory locations.

**Problem 1:** Write the table above with the addresses represented in hexadecimal. (Leave the values in binary.)

---

<sup>5</sup>Actually, sometimes computers store addresses as values, that is, as the content in a memory location. For example, languages such as C and C++ allow *pointer variables*. A pointer variable "points to" another variable by storing that other variable's address as the pointer variable's value. Pointers are one of the most confusing and error-prone part of programming. Because of this some programming languages do not include them, or severely limit their use.

We'll use hexadecimal representation of the addresses from now on. So a computer might assign location `88f0` to hold the value of `lineSum`. Does this mean it assigns location `88f1` to hold the value of `lineSumAug`? Recall from the last chapter that different types of data take different amounts of space. Let's suppose that both line counts are represented in the program as 16-bit integers. Location `88f0` (and all other locations) only hold 8 bits. So `lineSum` needs not only location `88f0`, but also `88f1`. And the computer needs to "remember" that `lineSum` uses 16-bits. That is why a variable specification needs to include its type.

Now let's look again at what happens, on this low level, for the following three lines of code:

```
lineSum = 1066
lineSumAug = lineSum
lineSum = lineSum + 10.
```

Assume when the computer system starts to run the program it sets aside memory locations `88f0` and `88f1` for `lineSum` and `88f2` and `88f3` for `lineSumAug`. Note  $1066_{10} = 0000010000101010_2$  and  $1076_{10} = 0000010000110100_2$ . Assume all these locations for these variables contain the value  $0000000000000000$  originally. Then assigning 1066 to `lineSum` changes the corresponding values:

Address	Value
88f0	00000100
88f1	00101010
88f2	00000000
88f3	00000000

Then the line `lineSumAug = lineSum` puts the value of the latter into the former:

Address	Value
88f0	00000100
88f1	00101010
88f2	00000100
88f3	00101010

Finally, adding 10 to `lineSum` changes that variable's value:

Address	Value
88f0	00000100
88f1	00110100
88f2	00000100
88f3	00101010

## 5.6 Machine Instructions

For an algorithm to run on a computer, someone needs to implement it as a computer program. However, computers need to process programs further before they can run them.

A program written in a high-level language such as Java or Python must first be converted to machine instructions. This is done by a special program called a compiler or interpreter, and produces a machine code or executable version of the program.<sup>6</sup>

Different types of computers have different instruction sets. An executable file for one type of computer will often not run on other types of computers. This is one reason why there are different versions of programs such as Microsoft Office for different types of machines.

Instruction sets consist of very low-level instructions. Some instructions are ones you would recognize, such as addition. Others are related to the inner workings of computers. For example, here is a simple set of about a dozen machine instructions for a single-register processor. Actual instruction sets are longer and more complicated, but this set will serve to illustrate the basics of machine code.

Op Code	Operation	Meaning
00000000	IN A	Have the user input a number; store it in address A
00000001	OUT A	Output the contents from address A in decimal
00000010	CLEAR A	Set the contents of address A to 0
00000011	STORE A	Copy the register contents to the contents of address A
00000100	LOAD A	Load the contents of address A into the register
00000101	INCREMENT A	Add 1 to the contents of address A
00000110	DECREMENT A	Subtract 1 from the contents of address A
00000111	ADD A	Add the contents of address A to the contents of the register and store the result in the register
00001000	COMPARE A	If the contents of address A are greater than the contents of the register, set the GT flag to 1, else to 0
00001001	JUMP L	Take the next instruction from address L
00001010	JUMPGT L	Take the next instruction from address L if the GT flag is 1
00001011	STOP	Stop program execution

When a program is turned into machine code, all the instructions in a high-level programming language are turned into the opcodes for machine instructions. For example, suppose we have a simple loop:

---

<sup>6</sup>This explanation is somewhat simplified as there are multiple details we are omitting. For example, the program code often needs to be linked with pre-existing library code.

```

Get i
Set sum to 0
While i is greater than 0
    Set sum to sum + i
    Set i to i - 1
Print sum

```

How would we turn this into machine instructions in terms of opcodes? Note for some lines there is an equivalent machine instruction:

Get i	IN I
Set sum to 0	CLEAR SUM
Set i to i - 1	DECREMENT I
Print sum	OUT SUM

However, other lines require more than one machine instruction. For example, `Set sum to sum + i` requires a sequence of three steps:

```

LOAD SUM
ADD I
STORE SUM

```

Moreover, both the beginning and end of the loop require care. For the beginning, we first want to check if `i` is greater than 0. This itself requires multiple steps: Clear a location so it holds the value 0. Then load that 0 value into the register. Next, compare the value of `i` to that 0 register value. If the value of `i` is greater than 0, then the GT flag (a special bit in the processor memory) is set to 1. We then check that bit and, if it is 1, jump to the first instruction in the loop; if it is 0, we jump to the first instruction past the loop. So the single `if` line yields the following, where `Z` corresponds to 0, and `L1` and `L2` are the memory addresses of the instructions `Set sum to sum + i` and `Print i`, respectively:<sup>7</sup>

```

CLEAR Z
LOAD Z
COMPARE I
JUMPGT L1
JUMP L2

```

After executing the line at the end of the loop, namely, `Set i to i-1`, we want to jump up to the start of the loop. This requires a `JUMP` to the address of the `CLEAR` instruction above. Finally, we also want a `STOP` at the end of the program.

Putting all these parts together produces the following, where we have given line numbers to jump destinations:

---

<sup>7</sup>Note this sequence is not unique (see if you can think of other ways to perform the same task but with a different sequence of instructions from the given machine instruction set). Moreover, actual machine instruction sets are larger, and would usually require fewer instructions for the loop's beginning line. The key point remains, however: one line of an algorithm, or one line of high-level programming code often corresponds to several lines of machine instructions.

```

    IN A
    CLEAR SUM
50   CLEAR Z
    LOAD Z
    COMPARE I
    JUMPGT 70
    JUMP 100
70   LOAD SUM
    ADD I
    STORE SUM
    DECREMENT I
    JUMP 50
100  OUT SUM
    STOP

```

This is an example of *assembly code*. However, it is still not binary. To change assembly code to machine code we need to perform two additional steps. First we replace operation names with binary opcodes. For example, IN becomes 00000000, CLEAR becomes 00000010, etc. Second, we replace all variables and line numbers with binary numbers representing memory locations. We can't choose arbitrary locations, but the operating system can assign line numbers based on where it places the machine code when that code is to be executed; and it can assign memory locations for the variables based on where there is convenient available memory. So, for example, if the system placed the first instruction at memory location 11001100, we could get the following table:

Memory Location	Op Code	Address	Assembly Code
11001100	00000000	11100100	IN I
11001110	00000010	11100110	CLEAR SUM
11001100	00000010	11101000	CLEAR Z
11001110	00000100	11101000	LOAD Z
11010000	00001000	11100100	COMPARE I
11010010	00001010	11010110	JUMPGT 70
11010100	00001001	11100000	JUMP 100
11010110	00000100	11100110	LOAD SUM
11011000	00000111	11100100	ADD I
11011010	00000011	11100110	STORE SUM
11011100	00000110	11100100	DECREMENT I
11011110	00001001	11001100	JUMP 50
11100000	00000001	11100110	OUT SUM
11100010	0001011		STOP
11100100			I
11100110			SUM
11101000			Z

Empty table locations are unused or will be filled during the program's execution. Note that most instruction lines involve an opcode and an associated address. The opcodes in this example take one byte; assume that line numbers and variables will also consist of a single byte value. So each line involves two bytes. Therefore the diagram shows two memory locations side by side in each row with only the address of the leftmost location of the pair being given.

To review, this table shows the machine code, the 0's and 1's, corresponding to the small piece of pseudocode at the start of this section. The leftmost column gives the memory addresses; these are not explicitly stored in the computer but are given here for reference. The second column from the left holds opcode values (except in the bottom three rows). These are the instructions that the computer will perform: to execute the program, the computer steps through each memory location containing the instructions, and performs the instruction associated with the opcode. As part of this, the computer usually needs an associated memory address; these are given in the next column over. Put another way, the contents of the third column are binary numbers that are stored in the computer; each of these numbers (except for the bottom four rows) provides a memory location for the operation given by the opcode immediately to its left. The bottom three rows are for variables in the program. The contents of the memory locations will be filled in, changed, and perhaps output as the program execution progresses.<sup>8</sup>

This example is admittedly complicated. Manually tracing through machine code or converting higher-level instructions into machine code is tedious and error-prone. That is why most programming is done in higher-level languages, with compilers or interpreters translating the high-level code to machine code.

## 5.7 Some Quantities

This section contains two tables. The first contains memory terms such as megabyte and gigabyte. These are important to remember since they come up frequently. It also contains some useful equivalents of memory amounts.<sup>9</sup> The second table contains prefixes for fractions of a second.

---

<sup>8</sup>To further stress the point that machine instructions and program variables are similar within a computer, i.e., all 0's and 1's, here is one type of computer security attack: The attacker feeds a program a much larger input string than the program expected. The program puts this string into memory, but because of its large size the input overflows its assigned memory locations and fills some memory after those locations. The overlong input contains some malicious code placed there by the attacker. The attacker's intent is for this code to overwrite part of a usual program and allow the attacker to gain control of the machine. Note this attack relies on being able to mix computer instructions and data.

<sup>9</sup>These equivalents are based on a similar table in G. Michael Schneider and Judith L. Gersting's *Invitation to Computer Science*.

Figure 5.1: Some Memory Equivalents: Approximation in Terms of Printed Material

Prefix	Approx. Amount	Equivalent
1 byte	8 bits	a single ASCII character
1 kilobyte (KB)	a thousand bytes	a single small page of text
1 megabyte (MB)	a million bytes	a long novel, or two or three shorter ones
1 gigabyte (GB)	a billion bytes	a large wallful of books
1 terabyte (TB)	a trillion bytes	a large library
1 petabyte (PB)	a quadrillion bytes	all libraries in the U.S.

Figure 5.2: Some Important Fractional Time Durations

Name	Duration	Duration (in words)
1 millisecond (ms)	$10^{-3}$ seconds	one-thousandth of a second
1 microsecond ( $\mu$ s)	$10^{-6}$ seconds	one-millionth of a second
1 nanosecond (ns)	$10^{-9}$ seconds	one-billionth of a second
1 picosecond (ps)	$10^{-12}$ seconds	one-trillionth of a second
1 femtosecond (fs)	$10^{-15}$ seconds	one-quadrillionth second

## 5.8 System Software

The sections above occasionally mentioned the operating system, compilers, and interpreters. These are examples of *system software*, that is, “helper” software whose purpose is to assist the computer in accomplishing other tasks. For example, compilers and interpreters turn programs in high-level languages to machine code. The operating system manages not only the running of programs, but also many other tasks the computer needs to perform to run smoothly. For example, you might be using a spreadsheet at the same time you are streaming music and downloading a data set from a web site. The computer needs to juggle all these tasks so that none interferes with the others.

We won’t have time to investigate system software in detail. Here are the key points you should know:

1. Not all programs have a user’s goal as their primary purpose. System programs’ purpose is to “make the computer work.”
2. Specific examples of system software include operating systems such as Linux, Windows, and MacOS. Other examples include compilers and interpreters such as the Python interpreter used in this class.

3. System software often performs invisible work: usually users don't even know a system program is running.
4. Some system software is large and complicated. For example, operating systems consist of millions of lines of code.
5. System programs are programs. So at the lowest level they are 0's and 1's as well.

## 5.9 Some Additional Notes

The computer model in this chapter is just one possible type of computer. There are other possible models. For example, *parallel computers* have a number of processors that work simultaneously. If a problem can be broken into independent or nearly independent subproblems, then different subproblems can be assigned to different processors, allowing the problem to be solved in less time.

Suppose a problem would normally take 500 minutes to solve. Suppose moreover that you have a 32-processor machine, the problem can be broken into 32 subproblems that can run simultaneously, and each subproblem can be solved in 1/32nd of the time of the original problem. Then each subproblem requires 500/32 minutes, or roughly 16 minutes. There would be some time breaking the original problem into subproblems, combining the subproblem answers for an overall answer, etc. But the key point of parallel computation is that some problems can be broken into subproblems that can be solved in parallel.<sup>10</sup>

Nowadays, many PCs are actually small multiprocessor machines. For example, a “quadcore” machine has four main processors. Remember, however, that this does not guarantee that all programs will run four times as fast: many programs cannot be easily broken into independent subparts.

There are still other computation models as well. For example, *quantum computing* is a current research topic. Quantum computers rely on an entirely different model of computation, and if they become a reality they will usher in significant changes in computer capabilities.

## 5.10 Further Questions

Here are some further questions to think about.

1. See if you can explain the basics of how computers work (or some part of how computers work, such as how different types of computer memory are organized) to someone who is unfamiliar with it.

---

<sup>10</sup>An alternative to parallel computation is *distributed computation*. Instead of having the subproblems solved on different processors on the same machine, distribute the subproblems to different single-processor machines. For example, you could distribute the example problem to a network of 32 PCs. The cost of many single-processor machines is usually less than that of a corresponding parallel computer. However, the data transmission times (over a network rather than within a machine) will be larger.

2. Think of another field you are interested in. Do low-level computer basics relate to this field in any way? For example, are there any special computational devices used in that field and, if so, how much memory do they have? Or does the field require regular computers with special capabilities such as enhanced graphics capabilities or extremely large amounts of memory?
3. Try to find out the computing characteristics of any computers or computational devices you have. For example, if you have a laptop, what is its processor speed? If you have an MP-3 player, how much memory does it have?
4. Think about how you use computers and what computer capabilities are most important. For example, if you work with large video files, large amounts of memory are important.

## 5.11 Additional Problems

**Problem 2:** Answer the following arithmetic problems on computer organization.

- (a) Suppose a computer has a maximum memory size of 4MB. What is the needed address field width?
- (b) Suppose a computer has an opcode field of 9 bits. What is the maximum number of instructions in its instruction set?
- (c) If a computer has an average memory access time of 10 nsec, and an average cache memory access time of 2 nsec, and the cache hit rate is 50%, then what is the overall average access time? A 50% cache hit rate means half the time the computer accesses the cache and finds the data; but the other half it accesses the cache, does not find the data, and therefore must get it from main memory.
- (d) Assume you have a list of 50 numbers, each of which takes 32 bits to store. The first number in the list is stored at memory address  $803762_{10}$ , and the numbers are stored consecutively in memory. What is the memory address of the first memory location after the last number in the list?

**Problem 3:** Answer the following questions related to memory size. Show your work.

- (a) The Apollo 11 spacecraft had 2KB of RAM.
  - (i) What is the minimum address field width needed for 2KB of memory?
  - (ii) Would the novel *Pride and Prejudice* fit into 2KB of memory?
  - (iii) Would a typical MP-3 file for a three (or so) minute song fit into 2KB of memory? (If you don't know the approximate length of an MP-3 file for a 3 minute song, look it up somewhere).
- (b) Suppose you have a computer with 1.2TB of free hard disk space.
  - (i) About how many images could you store in 1.2TB if the average image size is 400KB?

(ii) About how many short videos could you store in this space if the average video file size is 25MB?

(iii) About how many movies could you store in this space if the average movie file size is 2GB?

**Problem 4:** (This problem is tedious. One point of doing it is to get a good understanding of why most programmers do not program in machine code or assembly code). Suppose you have the following 4-bit instruction set. Note this is a different instruction set than in the example above. Below, ‘R’ indicates the register.

Op Code	Operation	Meaning
0000	CLEAR A	Set the contents of address A to 0
0001	STORE A	Copy the contents of R to the contents of address A
0010	LOAD A	Load the contents of address A into R
0011	INCREMENT A	Add 1 to the contents of address A
0100	DECREMENT A	Subtract 1 from the contents of address A
0101	ADD A	Add the contents of address A to the contents of R and store the result in R
0110	COMPARE A	If the contents of address A are greater than the contents of R, set the GT flag to 1, else to 0. If the contents are equal, set the EQ flag to 1, else to 0.
0111	JUMP L	Take the next instruction from address L
1000	JUMPGT L	Take the next instruction from address L if the GT flag is 1
1001	JUMPEQ L	Take the next instruction from address L if the EQ flag is 1
1010	IN A	Have the user input a number; store it in address A
1011	OUT A	Output the contents from address A in decimal
1100	STOP	Stop program execution
1101	unused	
1110	unused	
1111	unused	

Suppose further that the machine you are using has a 4-bit address field width. Note that in this problem a machine instruction and its associated memory location require 8 bits total: 4 for the instruction, and 4 for the location. So an instruction-address pair requires a single byte, in contrast to the 2 bytes in the example above.

Now suppose the program counter contains the value 0000 — i.e., it is pointing to the very first memory location — and the memory contents are as shown in the table below. This problem asks you to trace through the machine code in the contents of that table. Note the content at location 0000 is 10101110. This value contains opcode 1010 (the leftmost 4 bits) which means ‘IN,’ and its address field value is 1110. So this instruction gets an input value and places it in memory location 1110. Assume all the variables in this program are 8-bit integers, so each variable takes one memory location. After completing

this operation, the program counter changes to 0001; the content there is 10101111. This instruction gets an 8-bit input value and places it in memory location 1111. Similarly, the subsequent memory content consists of opcodes for additional instructions and their related memory locations.

Memory Address	Content
0000	10101110
0001	10101111
0010	00101110
0011	01101111
0100	10001001
0101	10011100
0110	10111110
0111	10111111
1000	01111101
1001	10111111
1010	10111110
1011	01111101
1100	10111111
1101	11000000
1110	00000000
1111	00000000

(a) What does the code in the table immediately above do if the input for the instruction at location 0000 is the value 5 (i.e,  $00000101_2$ ) and the input for the instruction at 0001 is the value 8 (i.e,  $00001000_2$ )? Trace through the machine code above starting at location 0000. For each executed instruction, state what it does. Then state what value or values, if anything, the program outputs.

(b) Do the same if the input for the instruction at location 0000 is  $140_{10}$  ( $10001100_2$ ) and the input for the instruction at 0001 is also  $140_{10}$ .

## 5.12 Problem Solutions

### Introductory Problem:

(a) Each record has a 6-character code, a 16-bit (or 2-byte) integer, and a 32-bit (or 4-byte) floating point number. Recall from the last chapter that in ASCII each character takes one byte, so the code takes 6 bytes. Therefore each record takes  $6 + 2 + 4 = 12$  bytes. Multiply this by the number of records, i.e., 100, to get 1200 bytes, or a little more than a kilobyte.

(b) The first record is at location  $3b2201aa_{16}$ . Note to get the start of the second record we add  $12_{10}$ , to get the start of the third record we add  $(2 \times 12)_{10}$ , etc. And so to get the start of the last, i.e., 100th, record we add  $(99 \times 12)_{10} = 1188_{10}$  to the first record's location.

This is somewhat tricky since the first location is in hexadecimal. Let's change part of it to decimal, add in decimal, then convert back to hexadecimal. It turns out we don't need to convert the entire hexadecimal location; we can just convert the last part since the number we are adding is relatively small, and there won't be any carrying that would affect the first part. So let's convert the last four digits,  $01aa_{16}$ . In decimal this equals  $0 \times 16^3 + 1 \times 16^2 + 10 \times 16^1 + 10 \times 16^0 = 256 + 160 + 10 = 426$ . Add 1188 to get 1614 in decimal. Now convert back to hexadecimal: note 1614 is less than  $16^3 = 4096$ , but  $16^2 = 256$  divides into it six times with a remainder of 78. Then  $16^1$  divides into 78 four times with a remainder of 14. So the last four digits of the hexadecimal location are  $064e$ , and the entire hexadecimal number for the last location is  $3b22064e$ .

Now we need to find the final memory location occupied by the 100th record. All records are 12 bytes long, and we just found that the first byte of the 100th record has location  $3b22064e$ . To find the address of the second byte of the record we add one to this starting address. To find the address of the third byte of the record we add two to the starting address. Etc. So to find the address of the final (12th) byte of the record, we add  $11_{10}$ .

You might be able to do this directly in hexadecimal. If not, you can convert the final two digits of the hexadecimal address  $3b22064e$  to decimal:  $4e_{16} = 4 \times 16^1 + 14 \times 16^0 = 78_{10}$ . Then add  $11_{10}$  bytes to get  $89_{10}$ . Then convert back to hexadecimal: 16 divides into 89 five times with a remainder of nine. So the address of the last byte of the last record is, in hexadecimal,  $3b220659$ .

**Problem 1:**

Address	Value
0000	00000000
0001	00000000
0002	00000000
0003	00000000
...	...
fffc	00000000
fffd	00000000
fffe	00000000
ffff	00000000

**Problem 2:**

(a) 22 bits since  $2^{22}$  bytes = 4MB.

(b) If the opcode field has  $n$  bits, then there are at most  $2^n$  possible instructions. So the maximum is  $2^9 = 512$  instructions.

(c)  $.5 \times 2 \text{ nsec} + .5 \times (10 + 2) \text{ nsec} = 7 \text{ nsec}$ .

(d) 32 bits is 4 bytes, and so the list takes  $4 \times 50 = 200$  bytes to store. So the list takes locations 803762 to 803961, and the next address is 803962.

**Problem 3:**

(a)(i) 11, since  $2^{11}$  bytes = 2KB.

(ii) No. 2KB is roughly enough space to store 2 small pages of text, not enough for an entire novel.

(iii) No. A typical MP-3 file for a 3 minute song is a few MB in size.

(b)(i) 1.2TB/400KB is approximately equal to 1,200,000,000,000 divided by 400,000, which equals 3 million.

(ii) 1.2TB/25MB is approximately equal to 1,200,000,000,000 divided by 25,000,000, which equals 48,000.

(ii) 1.2TB/2GB is approximately equal to 1,200,000,000,000 divided by 2,000,000,000, which equals 600.

**Problem 4:**

The opcode sequence is equivalent (but not line by line) to the following pseudocode. To make the code clearer, we'll call the content of memory location 1110 'X', and of 1111 'Y'.

```

1  Get X
2  Get Y
3  If Y < X then
4      Print X
5      Print Y
6  Else if Y > X
7      Print Y
8      Print X
9  Else
10     Print Y
11  Stop
```

(a) For input  $X = 5$  and  $Y = 8$ , the code outputs 8 followed by 5. A detailed explanation is in the first table below. As above, the table calls the content of memory location 1110 'X', and of 1111 'Y'.

(b) For input  $X = 140$  and  $Y = 140$  the code outputs 140, printed only once. A detailed explanation is in the second table below. As above, the table calls the content of memory location 1110 'X', and of 1111 'Y'.

Memory Address	Content	
0000	10101110	Gets value of 5 and stores it in X
0001	10101111	Gets value of 8 and stores it in Y
0010	00101110	Load the value of X (5) into R
0011	01101111	Compares value of Y (8) with value in R (5); Sets GT flag to 1
0100	10001001	Jumps to instruction at location 1001
0101	10011100	Not executed
0110	10111110	Not executed
0111	10111111	Not executed
1000	01111101	Not executed
1001	10111111	Outputs value of Y (8)
1010	10111110	Outputs value of X (5)
1011	01111101	Jump to instruction at location 1101
1100	10111111	Not executed
1101	11000000	Stop
1110	00000000	
1111	00000000	

Memory Address	Content	
0000	10101110	Gets value of 140 and stores it in X
0001	10101111	Gets value of 140 and stores it in Y
0010	00101110	Load the value of X (140) into R
0011	01101111	Compares value of Y (140) with value in R (140); Sets EQ flag to 1
0100	10001001	Checks GT flag but does not jump
0101	10011100	Checks EQ flag and jumps to address 1100
0110	10111110	Not executed
0111	10111111	Not executed
1000	01111101	Not executed
1001	10111111	Not executed
1010	10111110	Not executed
1011	01111101	Not executed
1100	10111111	Prints value of Y (140)
1101	11000000	Stop
1110	00000000	
1111	00000000	

## 5.13 Additional Resources

Here are additional resources. The first three items appeared above in the “Chapter Structure” section and are required reading. See that section for more information.

- [http://en.wikibooks.org/wiki/Computers\\_for\\_Beginners/Buying\\_A\\_Computer](http://en.wikibooks.org/wiki/Computers_for_Beginners/Buying_A_Computer). Wikibooks page on buying a computer. This provides background on different parts of computers, as well as the basic decisions you need to make when buying a computer.
- <http://computer.howstuffworks.com/computer-memory1.htm>. This is the *HowStuffWorks* “How Computer Memory Works” page. This page gives details about computer memory. Specifically, it covers many of the basics mentioned in class, as well as providing additional detail for people who’d like more information.
- <http://computer.howstuffworks.com/microprocessor.htm>. This is the *HowStuffWorks* “How Microprocessors work” page. Like the item above, this page cover many of the basics mentioned in class, as well as providing additional details for those who are interested.
- <http://en.wikipedia.org/wiki/Megabyte>. Wikipedia megabyte page. This page also contains a chart with different byte multiples.
- [http://www.simetric.co.uk/si\\_time.htm](http://www.simetric.co.uk/si_time.htm). Table with fractional seconds. This page also contains useful conversions such as the number of seconds in a year.



# Chapter 6

## Moore's Law

Faster, smaller, cheaper.

### 6.1 Introduction

#### 6.1.1 Introductory Problems

**Problem 1:** Suppose you have the choice of the following means of compensation: Alternative 1 gives you \$100 daily for 12 days. Alternative 2 gives you \$1 on the first day, \$2 on the second, \$4 on the third, \$8 on the fourth, etc. for 12 days, with each day after the first doubling the previous day's amount. Which alternative would give you more pay?

**Problem 2:** Suppose a certain type of computer has 40MB of memory currently, and the amount of memory is likely to double every two years for the next decade. How much memory is the computer likely to have 10 years from now?

#### 6.1.2 Introductory Explanation

“When the group moved to California to become part of Lucasfilm, we got close to making a computer-animated movie ... But when it came time to harden the deal and run the numbers for the contracts, I discovered to my dismay that computers were still too slow: The projected production cost was too high and the computation time way too long. We had to back out of the deal. This time, we did know enough detail to correctly apply Moore's Law and it told us that we had to wait another five years to start making the first movie. And sure enough, five years later Disney approached us to make *Toy Story*.

“Moore's Law told us that the new company we were starting, Pixar, had to bide its time ... ”<sup>1</sup>

---

<sup>1</sup>From “How Pixar Used Moore's Law to Predict the Future”, Alvy Ray Smith, in *Wired* magazine,

In the early days of computing — the 1950's and 1960's — computers were large objects that often filled a good part of a room. Today cell phones or calculators contain more computing capability than these large early machines. In fact the computing capability of a cell phone, for example, dwarfs that of a personal computer from just a couple decades ago.

This fast change is exhilarating, but also challenging. How can individuals, businesses, and other organizations keep up with this rapid pace of computing change? On one hand, the answer is they cannot — it is impossible to stay current with all computer advances, to always use the latest hardware and software, and to always predict what the next round of advances will be.

On the other hand, there are some tools that are useful both for understanding past advances in computing, and for trying to predict future advances. One of the most important of these tools is *Moore's Law*: that the density of transistors on a computer chip will double every two years. This is important because higher transistor density leads to computational devices that are smaller (in size), less costly, and more powerful in terms of computational ability.

Moore's Law is not a law in the same sense as a physical law — say Newton's Laws of Motion — or in the legal sense. Instead, it is an *observation* or *model*.<sup>2</sup> In particular, although there is nothing that *required* Moore's Law to hold in the past, it has been remarkably accurate in describing past advances in transistor density. And although there is nothing that *guarantees* Moore's Law will hold in the future, it is nonetheless a useful tool for predicting possible future advances. In short, Moore's Law is both *descriptive* of past computer advances and *predictive* of likely future advances.

One often hears variants of Moore's Law. For example, sometimes the time period is said to be every 18 months. Sometimes the law is stated that computing power or memory will double every two years. These variants aren't quite accurate; at the same time they aren't all wrong either. For example, while computer speed is not directly proportional to transistor density, it is related.

Doubling every two years leads to a type of increase called *exponential growth*, a term that is sometimes misused to mean rapid growth of any kind. We'll look at an exact definition of exponential growth in a section below. Roughly speaking, though, it means what you are measuring follows a formula like

$$\text{count at time } t = 100 \times 2^t.$$

Notice how the count progresses as  $t$  increases: when  $t = 0$  the count is 100, when  $t = 1$  the count is 200, when  $t$  is 2, the count is 400, and so on, with the count doubling each time  $t$  increases by 1.

---

April 17, 2013; available online at

<http://www.wired.com/opinion/2013/04/how-pixar-used-moores-law-to-predict-the-future/>

<sup>2</sup>One reason for the importance of mathematics in general is its effectiveness in modeling many phenomena. *Mathematical modeling* is the subarea of mathematics that deals with questions “What different types of mathematical models are there?”, “How do we measure ‘how good’ a model is?”, “How do we predict how well a given time-dependent model will hold in the future?”, etc.

Figure 6.1: A function that doubles every two time periods

$t$	$t/2$	$2^{t/2}$
0	0	1
2	1	2
4	2	4
6	3	8
8	4	16
10	5	32
12	6	64

To better understand Moore's Law, let's look at a function that doubles every two years, namely  $f(t) = 2^{t/2}$ . Some values for the function are shown in Table 6.1. The pattern in that table is, of course, a pattern you have seen before since similar patterns arose when we explored machine organization and data representation. In particular, note the rapid growth of the function values, a growth whose importance we will explore further in the next section.

### 6.1.3 Why is Moore's Law Important?

Moore's Law is important to people who work with computers for the reasons mentioned in the last section. First, since it has modeled the increase in transistor density so well, it is an explanatory tool for the rapid increases in computers. Second, it is also a tool for predicting advances in computing in the future.

But why is Moore's Law important in this class? Why should it be important to you? Here are a few reasons.

First, one goal in this class is to understand both computers and how computer scientists "see the world." Moore's Law is a powerful tool for computer scientists and engineers as they understand, discuss, and analyze computers and computing.

Second, Moore's Law helps us appreciate the rapid advances that have occurred in computer technology. For example, in the early days of computing, computers were large machines that filled the better part of a room, and were so expensive that they were not owned by individuals but only by government organizations, large businesses, or research universities. Nowadays the computational abilities of a cell phone dwarf the computational abilities of those early computers.

Third, Moore's Law helps us understand the effect these rapid advances in computers have had on society. The fact that computers are becoming more powerful, smaller, and cheaper has many societal ramifications, some of which we'll explore elsewhere in this class.

Fourth, Moore's Law furnishes a tool for predicting future changes in computing, and

its possible effects on society. Like Pixar in the opening quote of this chapter, we might be interested in predicting when the time will be right for a certain application.

Finally, Moore's Law is an example of exponential growth. This is a very important idea, not only in computer science, but also in other areas such as biology, physics, and finance. It is not uncommon to hear that something is "growing exponentially." Understanding what this means is important in understanding many issues affecting individuals and society.

In summary, Moore's Law helps us understand, explain, and analyze

- computers and their underpinnings;
- the rapid advances in computer technology;
- the many societal effects of these rapid advances;
- possible future changes in computing technology, and the effects of those changes on society; and
- exponential growth.

**Problem 3:** Why specifically might Moore's Law be interesting to you? Give specifics about why any of the reasons mentioned are particularly important to you, or give other reasons why Moore's Law might be interesting.

### 6.1.4 Topic Goals

Upon completing this topic, you should be able to do the following.

- Explain what Moore's Law is and why it is important.
- Be able to use Moore's Law to explain some past advances in computing, and analyze some predictions about computing's future.
- Be able to explain what exponential growth is, and why it is important.
- Be able to find characteristics of, and do computations with, functions that exhibit exponential growth.

### 6.1.5 Moore's Law and the Liberal Education Requirements

Moore's Law is related to both liberal education requirements in this class: it is a mathematical model of a very important aspect of computer hardware, namely transistor density. And this model involves exponential growth, which is an important concept in many areas, not just computer hardware or computer science.

Both the descriptive and predictive features of Moore's Law also are useful in thinking about technology and society. As mentioned in the previous sections, Moore's Law is useful in understanding, discussing, and analyzing past effects of computers on society, and in predicting future effects.

## 6.2 More About Exponential Growth

What exactly does “exponential growth” mean? As mentioned above, it commonly is used to denote rapid growth. But technically speaking it means the growth of functions of the form

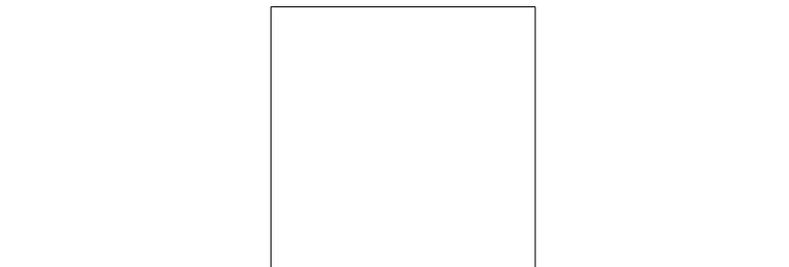
$$f(t) = a \times b^t$$

where  $b > 1.0$ . This might seem very abstract, but it has some easy to understand and important characteristics.

First, notice that at time  $t = 0$  the function has value  $a \times b^0 = a \times 1 = a$ . So the constant  $a$  gives an initial value of whatever we are counting. Second, notice that whenever we add 1 to  $t$  — regardless of what the actual value of  $t$  is — the function value increases by a factor of  $b$  since  $f(t+1)/f(t) = (a \times b^{t+1})/(a \times b^t) = b^{t+1}/b^t = b$ .

To make this more concrete, consider a drawing design problem. Suppose you have a square. You replace the middle third of each edge of the square with three line segments to get the more complicated edge shape shown in Figure 6.2. This turns each edge into

Figure 6.2: One side’s segment(s) after replacing the middle third



a sequence of five line segments. Since you do this for each of the four original edges of the square, the total number of segments becomes 20. You repeat this process a second time. Now the 20 become 100. Doing this a third time yields 500.

**Problem 4:** What is the function giving the line segment count after  $t$  applications of this process?

It is sometimes useful to write an exponential growth function in the slightly different form  $a \times b^{rt}$ . For example, the function  $f(t) = 2^{t/2}$  is in this form. We could write this function as  $f(t) = (\sqrt{2})^t$  — this is the exact same function — but the form  $f(t) = 2^{t/2}$  is often more informative. Notice that in this case  $r = 1/2$ ; this corresponds to a doubling period of two times steps. If  $r = 1/5$  the function is  $2^{t/5}$ , which doubles every five time steps. If  $r = 3$  the doubling period would be every  $1/3$  time steps. In general the doubling period is  $1/r$  time steps.<sup>3</sup>

<sup>3</sup>Notice how  $r$  affects the yearly rate of increase. The ratio of the count from time  $t+1$  to time  $t$  is  $f(t+1)/f(t) = (a \times b^{r(t+1)})/(a \times b^{rt}) = b^r$ . So, for example, the *yearly* rate of increase for the function  $f(t) = 2^{t/2}$  is  $2^{1/2} = \sqrt{2} \approx 1.4$ .

If we replace the 2 with a 3, i.e., consider  $f = 3^{rt}$ , then similar observations hold for the *tripling* period. For example if  $r = 1/2$  then the function will triple every two time steps.<sup>4</sup>

**Problem 5:** Consider the function  $f(t) = 200 \times 4^t$ . What is its initial value, i.e., its value when  $t = 0$ ? By what factor does it increase every time period?

As mentioned above, people sometimes mistakenly call any type of rapid growth exponential growth. To explore this further, suppose you find that the number of dwellings in a city has roughly followed the function  $g(t) = 200 + 100t^2$ . Note that  $g(0) = 200$ ,  $g(1) = 300$ ,  $g(2) = 600$ ,  $g(3) = 1100$ ,  $g(4) = 1800$  and so on. This function does grow quickly. Moreover it is greater than the function  $f(t) = 2^{t/2}$  for small values of  $t$ . But the growth of  $g$  is not exponential:  $g$  does not have the exponential growth form. This in turn means that

- unlike an exponential function, the ratio  $g(t+1)/g(t)$  is not constant, but actually decreases as  $t$  gets larger;
- when  $t$  gets sufficiently large, any exponential growth function will be larger and grow faster than  $g$ .

As a final note, while most functions that grow exponentially also grow rapidly, that is not always the case. For example, if you have an initial investment of \$1,000 that is growing at 4% compounded annually, then your amount after  $t$  years is  $1000 \times 1.04^t$ . Note that each year this function increases by a factor of 1.04. If you are familiar with compound interest you know that the compounding — the exponential growth ratio of 1.04 — has a powerful cumulative effect. However, is this rapid growth? The answer to that is context dependent. A growth rate of 1.04 is significantly more than, say, a growth rate of 1.01. However, it is significantly less than the rapid rate of a function such as  $2^t$ .

**Problem 6:** Compare the following functions:  $2^{t/2}$ ,  $1000 \times t^3$ ,  $2^t$ ,  $100 \times 1.05^t$ ,  $3^t$ . Which is largest when  $t = 0$ ? Which grows more quickly and is largest once  $t$  is large?

### 6.3 How Long Will Moore's Law Continue to Hold?

Moore's Law cannot hold forever. Yet Moore's Law has continued to model transistor density accurately, even in the face of frequent predictions of its imminent demise. Part of the story of Moore's Law has been how chemists, material scientists, electrical engineers, computer engineers, etc. have overcome seemingly insurmountable technical difficulty after technical difficulty to continually improve computer chip design and manufacture.

However, at some point the technical difficulties will be too much.<sup>5</sup> Similar to a business whose profits are doubling every year, or an app whose number of users is growing

---

<sup>4</sup>If the constant  $r$  is negative, then the count is actually decreasing. *Exponential decay* is an important concept in mathematics and is used to model phenomena such as radioactive decay, or certain drug effects in pharmacology.

<sup>5</sup>Some observers have already claimed that the doubling period has slowed recently.

by a factor of 10 every quarter (of a year), there will be a time at which that growth rate will no longer hold. Perhaps that time will come soon; perhaps it will not come for many years. Perhaps it will come abruptly; perhaps it will arrive gradually. Or perhaps there will be a revolution, such as quantum computing, that will cause a discontinuous leap in computer capabilities, and will require a different model of computation. It is difficult to predict what the future will hold. However, it is likely that computer capabilities will continue to increase significantly and rapidly in the future.

## 6.4 Questions

Here are some additional questions to think about. We will discuss some of these in class.

1. Explain in your own words what Moore's Law is and why it is important.
2. Explain in your own words what exponential growth is, and why it is important.
3. Choose an area (other than computer science) you are interested in. Can you think of any examples of exponential growth in that area?

## 6.5 Problem Solutions

**Problem 1:** Alternative 1 gives you  $12 \times \$100 = \$1200$ . Alternative 2 gives you  $1 + 2 + 4 + \dots + 2^{11} = \$4095$ . So Alternative 2 gives more total compensation.

**Problem 2:** The amount of memory will double five times in 10 years, so the amount will be  $40 \times 2^5 = 40 \times 32 = 1280$  MB or about 1.28GB.

**Problem 3:** Answers will vary.

**Problem 4:**  $4 \times 5^t$ .

**Problem 5:** The initial value is  $f(0) = 200$ . Each time period the function increases by a factor of 4.

**Problem 6:** When  $t = 0$  we have  $1000 \times t^3$  equals 1000,  $100 \times 1.05^t$  equals 100, and all the other functions are 1. Here are the functions in terms of increasing order of growth as  $t$  gets large:  $1000 \times t^3$ ,  $100 \times 1.05^t$ ,  $2^{t/2}$ ,  $2^t$ ,  $3^t$ .

## 6.6 Additional Resources

Here are some online resources on Moore's Law and exponential growth:

- [http://news.cnet.com/FAQ-Forty-years-of-Moores-Law/2100-1006\\_3-5647824.html](http://news.cnet.com/FAQ-Forty-years-of-Moores-Law/2100-1006_3-5647824.html). "FAQ: Forty Years of Moore's Law". This *CNET* article, from 2005, explains the basics of Moore's Law.

- <http://www.wired.com/opinion/2013/04/how-pixar-used-moores-law-to-predict-the-future/> “How Pixar Used Moore’s Law to Predict the Future”. This *Wired* magazine article provides an example of how researchers and businesses use Moore’s Law to predict future computing capabilities.
- [http://en.wikipedia.org/wiki/Moore's\\_law](http://en.wikipedia.org/wiki/Moore's_law). This *Wikipedia* page on Moore’s Law contains both some introductory material, as well as a number of interesting details, related information, and some advanced material.
- [http://en.wikipedia.org/wiki/Exponential\\_growth](http://en.wikipedia.org/wiki/Exponential_growth). This *Wikipedia* page on exponential growth provides a number of examples, as well as some additional (mostly advanced) material.
- <http://computer.howstuffworks.com/moores-law.htm>. This *HowStuffWorks* article provides background information about Moore’s Law. It is recommended for students who want to know more background than we will cover in class.

# Chapter 7

## Computer Security

Is security computing’s “Achilles’ heel?”<sup>1</sup>

### 7.1 Introduction

#### 7.1.1 Introductory Problem

The alphabetic string UGG’N QBGG OEUO BJAN QBGG was obtained by encrypting a well-known saying using an *alphabetic substitution cipher*. That is, each letter in the original saying was replaced by another letter. If a letter appeared more than once in the original saying, it was always replaced by the same letter. What is the original saying?

#### 7.1.2 Overview

Computer security is an important and complicated topic. It affects individuals, groups, companies, and governments. Because of the breadth and depth of this topic, we will not even come close to covering it in full. Instead we’ll look at at questions such as

- What is “computer security” and why is it important?
- What are some of the issues in computer security?
- How do encryption and decryption work?
- What are the most important things to know about computer security?

Some of these issues are addressed in this chapter. Others we’ll explore further during class time.

---

<sup>1</sup>“*Achilles’ heel*: a fault or weakness that causes or could cause someone or something to fail.” From the Merriam-Webster online dictionary <http://www.merriam-webster.com/>. Accessed Nov. 21, 2013.

### 7.1.3 Topic Goals

Upon completing this section, you should be able to do the following:

1. Be able to explain what computer security is and why it is important.
2. Be able to explain some of the important issues in computer security.
3. Be able to explain the basics of cryptography, and solve (not extremely complicated) encryption and decryption problems.
4. Be able to explain the basics of public key encryption: how it works (on a high level), what its advantages and disadvantages are, and what types of attacks it protects against.

### 7.1.4 How These Topics Relate to Mathematics

Mathematics provides many underpinnings of encoding and decoding messages securely. This is particularly true for strong encryption and decryption techniques, the types used in e-commerce or by the military. (As an example, see the section on public key encryption.) However, it is also true on a simpler level for less secure encryption or decryption techniques, such as those used in recreational puzzles.

### 7.1.5 How These Topics Relate to Society and Technology

Computer security is obviously an important topic for society. A need for security underlies individual computer use: we do not want unauthorized people accessing our personal computing devices, for example. It underlies business use of computers: businesses need to protect their transactions, data, etc. Without sufficient security, e-commerce would not exist. It underlies many “critical industries,” for example computers play a role in areas such as telecommunications, manufacturing, and energy. These industries obviously have strong reasons to keep their systems and facilities secure. And it underlies government functions, not only in areas like national security and the military, but also in areas such as government-run medical programs, tax records and collection, and law enforcement.

## 7.2 Some Computer Security Principles

We’ll start by making some “big picture” observations about computer security.

1. *Cryptography is a part, but not the entirety, of computer security.* Sometimes computer security is simplified to “encrypt your data and your communications.” While encryption is part of computer security, it is not the whole of it; there are many, many important aspects of computer security other than encryption.

2. *There are a variety of computer security attacks.* For example, some attacks target data, some transmission of data, some computer systems. Some attacks target individuals' personal computers. Some target business or government systems. Some attacks attempt to "steal data" but not to otherwise compromise a system. Some attempt to damage the system or render it ineffective for a period of time.
3. *Computer security has both technical and social aspects.* Computer security guidelines often focus on technical aspects: make sure your anti-malware software is up to date, use encryption on sensitive data, use sufficiently long passwords, etc. However, there are non-technical attacks as well. For example, one *social engineering* attack against a business involves a malicious hacker calling an employee and impersonating someone of importance within the business; for instance the hacker might pretend to be a technically inept boss with urgent need for a certain file.
4. *Computer security is not restricted to preventing unauthorized access by "outsiders."* While this is one type of threat, a large number of security incidents involve insiders. That is why a good organizational security plan involves items such as background checks for prospective employees, training about security practices and rules, restricting access to systems and data to only certain employees, logs of all activity on critical systems, etc.
5. *If you work for an organization that deals with sensitive data, there are probably developed practices and procedures you will need to follow.* For example, there are federal laws such as FERPA (that deals with educational data) and HIPAA (that deals with medical data) that employees must know and follow. Moreover, many organizations have their own internal security practices and rules.
6. *"Security is a process, not a product."* This principle, attributed to computer security expert Bruce Schneier, highlights that security isn't something you can get or buy and then be done with. For example, as important as anti-malware software is, it is only part of a good security plan. Additionally, new security threats are emerging on a weekly basis. What works today might not work tomorrow.
7. *It is useful to know some of the common types of attacks, and to know the basics of what individuals should do to protect themselves.* This includes being able to recognize some of the most common types of attacks (e.g., emails or phone calls that ask you to provide your password), and being familiar with some of the most common countermeasures (such as anti-malware software).
8. *There are a number of topics such as privacy that are related to security, but are not identical.* For example, sometimes security and privacy go hand-in-hand: an illegal access to a medical database is both a security and a privacy breach. Sometimes, though, more security means less privacy. For example, companies or agencies that work with critical systems often do extensive background checks on their prospective

employees; the result is more security for the company or agency, but less privacy for the prospective employee.

9. *Computer security is an issue of strong current concern.* In general, our computer systems are not as secure as they should be. However, making systems more secure is difficult for a number of reasons. These include that there are a number of different types of attacks; that many systems were developed before security became a concern, and it is difficult to add security to existing systems; that security has both technical and non-technical aspects; and that good security is costly.

### 7.3 Introduction to Cryptography

When you log into a computer system remotely, you don't want your password to be transmitted over a computer network in its usual form: it is too easy for someone to snoop on Internet traffic and discover your password. Instead, you want it to be encrypted, and any reasonable computer system encrypts passwords. Similarly, you don't want to send your credit card information unencrypted over the Internet, you don't want your bank doing financial transactions insecurely, and you don't want sensitive medical information stored where anyone can access it. For example, suppose a medical worker has a laptop containing medical records. Laptops are often stolen or lost, so extremely sensitive data should not be stored on them. But if it is, then it should be encrypted to make it more difficult for a thief to use it maliciously.

In this course we'll look in particular at encryption/decryption. This is not because this is the only important subarea of computer security; as the last section mentioned, it is only one of many. But it is an important subarea, and has a nice connection to the mathematical requirements of our course.

Cryptography, the science and practice of encrypting information,<sup>2</sup> is a complicated area. Large financial transactions, military communications, etc. use extremely sophisticated encryption techniques, which are beyond the scope of this chapter. So here we will focus on some of the basics.

Suppose Bob wants to send a message to Alice. However, Bob and Alice's archnemesis, Charlie, wants to intercept this message. How can Bob send Alice a message securely? This involves a number of steps if Bob and Alice use encryption:

1. Bob must take the original message, called the *plaintext*, then encrypt it using an *encryption key* or *cipher*. The result is called *ciphertext*.
2. Bob then sends the ciphertext to Alice.

---

<sup>2</sup>There is a large amount of special terminology in the area of computer security. Since this is an introductory class we will, for the most part, avoid much of the terminology. However, if you are interested, there are a number of online glossaries.

3. Alice receives the message, then applies a *decryption key* to the ciphertext to recover the original plaintext.

For this to work, Alice must have a decryption key that reverses what Bob's encryption key does. For example, if Bob's key advances each letter three places forward in the alphabet, then Alice's key would move each letter back three places.<sup>3</sup> Or if Bob's key is an alphabetic substitution cipher and it takes 'A' to 'K', then the decryption key would take 'K' to 'A'.

This presents a problem, however: how does Bob let Alice know what the key is without letting Charlie also know? If Bob and Alice are in different locations, having Bob send the key introduces the same problem as sending the original plaintext without encrypting it: Charlie might intercept the message. This is a problem with traditional cryptography: how to get the key securely to the receiver?

Note Charlie's aim is likely to discover what message Bob is sending to Alice. But this is not the only possibility. He might prevent the message from getting to Alice by stopping the transmission somehow. He might change the message to say something different, for example, instead of "Bid up to \$2,000,000 for the rare manuscript" he might change it to "Bid up to \$20,000,000 for the rare manuscript."

You have probably played games similar to the introductory problem in this chapter. These games might involve simple encryption techniques such as transposing pairs of adjacent letters, or advancing letters forward in the alphabet a set number of places. For example, suppose you have the message ARRIVE IN DUBLIN and move each letter forward 13 places in the alphabet (wrapping around if you go beyond the end). Just encrypt the letters; leave the spaces as they are.<sup>4</sup> What do you get? The answer is at the end of the next paragraph.

This method takes 'A' to 'N', 'R' to 'E', etc. This is an example of an *alphabetic rotation* cipher. This particular one, advancing every character 13 places, is called *ROT-13*. It is often used in non-secure settings, for example, to encrypt a "spoiler" comment in an online discussion of books or movies. Note that to decrypt ROT-13 ciphertext we merely shift all characters back 13 characters. However, also note we can advance them 13 characters; that is, since our alphabet has 26 characters, the ROT-13 encryption key is identical to the ROT-13 decryption key. (The answer to the problem above is NEEVIR VA QHOYVA. )

**Problem 1:** Suppose you know the ciphertext SWPAN WJZ NQOP was obtained through an alphabetic rotation technique, but don't know how many places each letter was advanced. What is the maximum number of possible rotations you'd need to try, and what is the plaintext?

The problem at the start of this chapter uses alphabetic substitution. Although significantly more complicated than alphabetic rotation, substitution is still not secure. Is

---

<sup>3</sup>This extremely simple technique is called the *Caesar cipher*.

<sup>4</sup>In serious encryption all characters including spaces and punctuation are encrypted so they do not provide clues to word length, sentence structure, etc.

this because alphabetic substitution does not provide enough possibilities? There are  $26 \times 25 \times 24 \times \dots \times 3 \times 2 \times 1$  different alphabetic substitution options. This is a very large number. The problem is not that there are so few possibilities that the computer can check all of them. Instead, the problem is that the computer can use other information. For example, certain letters appear more frequently than others. If you are familiar with the board game Scrabble, think about the low scoring letters such as ‘E’, ‘N’, and ‘S’. These letters appear much more frequently than the high-scoring letters such as ‘Z’, ‘Q’, and ‘J’. This frequency information suggests some likely correspondences, which an analyst can use to decrypt ciphertext obtained through alphabetical substitution.

In summary, alphabetic substitution and other simple techniques are not secure.

## 7.4 Public Key Encryption

One problem mentioned above is the “key distribution” problem: how does Bob securely send Alice a key so that she can decrypt messages that he sends? There is an ingenious technique called *public key encryption* that avoids the key distribution problem.

Public key encryption relies on the idea of irreversible (or, more accurately, difficult to reverse) processes. Some things are much easier to do than undo. In particular, it is easier to do mathematical operations such as multiplying two large numbers than the reverse operation: given a large number, factor it.

Public key encryption relies on this in producing public key/private key pairs. If Bob and Alice plan to use public key encryption, then Alice would first need to use a program to generate a public and a private key. She can “publish” the public key, so Bob can get it. However, only she should know the private key. Bob and Alice then do the following steps:

1. Bob encrypts the message using Alice’s *public* key.
2. Bob send the ciphertext to Alice.
3. Alice decrypts the message using her *private* key.

Note that anyone (including Bob) can send Alice a message using her public key. However, only she should be able to decrypt those messages.

As an alternative, sometimes public key encryption is used to send another key. This still avoids the key distribution problem, but allows using a less computationally intensive key for encrypting and decrypting long messages. So in this case there are two sets of encryption and decryption keys — the public key encryption and decryption keys, and the “regular” encryption and decryption keys. The steps are then as follows:

1. Bob encrypts the regular decryption key using Alice’s public key.
2. Bob sends the ciphertext version of the regular decryption key to Alice.

3. Alice decrypts the regular decryption key using her private key.
4. Bob encrypts a message using the regular encryption key.
5. Bob send the ciphertext version of the message to Alice.
6. Alice decrypts the message using the regular decryption key.

**Problem 2:** Is public key encryption foolproof? Does it defend against all types of attacks?

Here is a further explanation of how public key encryption works, using a type called RSA encryption. It relies on a number of mathematical concepts:

- *Prime numbers:* Prime numbers are positive integers that are greater than 1, and that are evenly divisible only by 1 and themselves. So, for example, 7 is prime since it has no divisors other than 1 and itself. However, 6 is *composite* (i.e., not prime) since it is also divisible by 2 and 3.
- *Greatest common divisors:* The greatest common divisor of two positive integers is the largest integer that evenly divides both. For example, the greatest common divisor of 30 and 36 is 6, since 6 divides both, and no larger integer does. The greatest common divisor is often abbreviated as *gcd*. For example,  $gcd(10, 15) = 5$ .
- *Relative primeness:* Two positive integers are said to be relatively prime if their greatest common divisor is 1, i.e., if no positive integer greater than 1 divides both evenly. For example, 9 and 14 are relatively prime, but 10 and 28 are not since 2 divides both.
- *Modular arithmetic:* The process below uses modular arithmetic, that is, the remainder operator. Specifically,  $a \bmod b$  is the remainder when you divide the integer  $a$  by the integer  $b$ . For example  $22 \bmod 10$  is 2.
- *Exponentiation:* The RSA process uses exponentiation extensively.

Here is an example. It is not important to follow all the details and arithmetic in the example. But you should get a feeling for the amount of computation involved, and how the mathematical items mentioned above play a role. This example illustrates the underlying mathematics; but in practice these steps are actually done by computer. So this is an illustrative example rather than what people actually do when using public key cryptography.

1. To generate a public/private key pair I take any two relatively large primes  $p$  and  $q$  (we'll use 1063 and 8501 for this example, although primes actually used are much larger) and multiply them together to get  $n = pq$ : 9036563.

2. I next take any number  $e$  with the property that  $\gcd(e, (p-1)(q-1)) = 1$  (e.g.,  $e = 43$  works since  $\gcd(43, (1063-1) \times (8501-1)) = 1$ ).
3. I send you  $e$  and  $n$ .
4. To send me an encrypted message you take your message, and first assign numerical equivalents to each character. There are many possible ways to do this; let's use the simple technique of assigning each letter its place in the alphabet: 'A' is 01, 'B' is 02, etc. Then group the digits into blocks of a certain size, say of length 4. Let  $M$  indicate a block of 4 digits. You then encrypt  $M$  by computing  $C = M^e \bmod n$ . You do this for each block, and send me the results. For example, if your original message is STOP, you'd have the following:

```

original message:      S   T   O   P
numerical equivalents: 18  19  14  15
you compute and send:
    1819 raised to the 43rd power mod 9036563 = 7757145
    1415 raised to the 43rd power mod 9036563 = 8393137

```

5. I know  $p$  and  $q$  (but  $n$  is, in practice, large enough that  $p$  and  $q$  cannot be found by other people). From these I find a number  $d$  such that  $de = 1 \bmod (p-1)(q-1)$ , or  $de = 1 \bmod 9027000$ . ( $d = 6297907$  works, we'll skip the mathematics involved in how to find  $d$ .) Apply this to  $C$ :  $C^d \bmod n = (M^e)^d \bmod n = M^{(de)} \bmod n$ . Now by a little mathematical "magic" this equals  $M \bmod n$ , and so I recover the original  $M$  by taking each block of what you sent me, and raising it to the  $d$  power mod  $n$ .

Comments:

- Note I can give you (and everyone else) the encryption key ( $e$  and  $n$ ) as long as  $n$  is difficult to factor.
- Note how prime numbers and the other mathematics mentioned above are involved in the encryption and decryption.
- The important thing with this example is not that you understand all the details, but that you understand the important points:
  - The motivation for public key encryption, as well as what problems it solves and what problems it doesn't.
  - How it works on a high level.
  - The reliance of public key encryption on mathematics.
  - The need for computers in doing the many computations involved in public key encryption.

## 7.5 Additional Problems

Here are some additional problems on this topic.

**Problem 3:** Do the following encryption/decryption problems manually (i.e., without relying on a computer). You need only encrypt the alphabetic characters — leave any punctuation as is.

(a) Encrypt the following Alice in Wonderland quote using ROT-13:

WHY IS A RAVEN LIKE A WRITING DESK?

(b) Decrypt the following message, which was encrypted using rotation-based encryption:

IJ RDNZ ADNC RJPGY BJ VITRCZMZ RDOCJPO V KJMKJDNZ

(c) Decrypt the following message, which was generated with substitution-based encryption:

MU BERZ, NEZE GE MVYW ZVP RY KRYW RY GE ARP IVYW WC YWRU XP THRAE.  
RPB XK UCV GXYN WC LC RPUGNEZE UCV MVYW ZVP WGXAE RY KRYW RY WNRW.

**Problem 4:** Algorithms such as public key encryption rely on being able to do “expmod” operations quickly for large numbers. That is, for given positive integers  $b, n, m$  they need to compute  $b^n \bmod m$ . Remember the mod  $m$  operations gives the remainder for division by  $m$ .

Computing  $b^n \bmod m$  by first computing  $b^n$  and then finding the remainder has two shortcomings. First, since  $b, n$ , and  $m$  are very large numbers,  $b^n$  is very, very large. And while computers can deal with very large numbers, it takes special efforts for them to do so. Second, this approach is rather time consuming.

Here is a much more efficient algorithm:

*Input:* Three positive integers  $b, n, m$ .

*Output:*  $b^n \bmod m$ .

- 1 Find the binary representation of  $n$ ; call the bits in it  $a_k a_{k-1} \dots a_2 a_1$
- 2 Set  $x = 1$
- 3 Set  $power = b \bmod m$
- 4 For  $i = 1$  to  $k$
- 5     If  $a_i$  equals 1
- 6         Set  $x = (x \cdot power) \bmod m$
- 7     Set  $power = (power \cdot power) \bmod m$
- 8 Print  $x$
- 9 Stop

(a) Trace through this algorithm for  $b = 7$ ,  $n = 17$ ,  $m = 13$ . Specifically, state (i) the binary representation of  $n$ , (ii) the values of  $x$  and  $power$  right after Line 3 is executed,

(iii) the values of  $i$ ,  $x$ , and  $power$  right after each time Line 7 is executed, and (iv) what value is printed.

(b) Trace through this algorithm for  $b = 5$ ,  $n = 23$ ,  $m = 11$ . Specifically, state (i) the binary representation of  $n$ , (ii) the values of  $x$  and  $power$  right after Line 3 is executed, (iii) the values of  $i$ ,  $x$  and  $power$  right after each time Line 7 is executed, and (iv) what value is printed.

**Problem 5:** Let's use another technique to encrypt (a shortened version of) the Alice in Wonderland quote from the problem above:

#### WHY A RAVEN?

(a) First, change each character, including the spaces and the question mark, into its decimal ASCII equivalent. Write all the decimal numbers together from left to right to form a single number.

(b) Take your answer from part (a) and transpose each pair of digits. So, for example, if your answer in part (a) was 12345678, you'd transpose the 1 and the 2, then the 3 and the 4, then the 5 and the 6, then the 7 and the 8, to get 21436587. (Your actual answer from part (a) should be longer than the 8-digit example used here.)

(c) Now move the last digit to the front of the number. Then taking two digits at a time, from left to right, turn each two-digit pair into the corresponding ASCII equivalent. So, for example, if you have 21436587 from part (b), moving the last digit to the front would give 72143658. Then 72 is an H, 14 is the non-printing SO (shift out) character, 36 is a dollar sign \$, and 58 is a colon : . In writing your final answer, enclose any non-printing characters in parentheses. So the encrypted message would be H(shift out)\$:

(d) Does this encryption result in a substitution key? That is, each time a specific character occurs in the plaintext is it encrypted as the same character in the ciphertext?

**Problem 6:** Are substitution-based ciphers secure? Briefly explain why or why not.

## 7.6 Problem Solutions

**Introductory Problem:** The saying is "ALL'S WELL THAT ENDS WELL."

**Problem 1:** There are 25 possible rotations to try (we don't count the 26th possibility, of advancing each letter 26 places, since it just replaces each letter with itself). The original message is WATER AND RUST.

**Problem 2:** It is not foolproof. For example, someone could pretend to be Bob and send a fake message to Alice.

**Problem 3** (a) JUL VF N ENIRA YVXR N JEVGVAT QRFV?

(b) NO WISE FISH WOULD GO ANYWHERE WITHOUT A PORPOISE

(c) MY DEAR, HERE WE MUST RUN AS FAST AS WE CAN JUST TO STAY IN PLACE. AND IF YOU WISH TO GO ANYWHERE YOU MUST RUN TWICE AS FAST AS THAT.

**Problem 4**

(a)

(i) 10001

(ii) x: 1 power: 7

(iii)	i	x	power
	---	---	-----
	1	7	10
	2	7	9
	3	7	3
	4	7	9
	5	11	3

(iv) 11

(b)

(i) 10111

(ii) x: 1 power: 5

(iii)	i	x	power
	---	---	-----
	1	5	3
	2	4	9
	3	3	4
	4	3	5
	5	4	3

(iv) 4

**Problem 5**

(a) 877289326532826586697863

(b) 782798235623285668968736

(c) Moving the last digit to the front gives 678279823562328566896873, which translates to CROR#>(space)UBYDI

(d) It isn't. For example, the fifth and eighth characters in the plaintext are the same — both A's — but the fifth and eighth characters in the ciphertext are not.

**Problem 6:** Substitution ciphers are not secure. This is because — even though there are an extremely large number of them — they can still be cracked by using techniques such as analyzing letter frequency and letter combination frequency.

## 7.7 Questions

Here are some questions to think about:

1. Suppose someone asked you for advice on personal computer security. List three tips you would give them.
2. There were security threats prior to computers. How have computers changed individual, organizational, and national security?
3. List some types of data that require strong security, and some types that require minimal, if any, security.
4. The issue of privacy is related to, but not identical to, security. How are computers changing the privacy landscape? List some ways or give some examples.

## 7.8 Additional Resources

Here are some additional, online resources:

- <http://computer.howstuffworks.com/encryption.htm>: the howstuffworks “How Encryption Works” page. This page covers many of the basics mentioned in class, as well as providing additional details for those who are interested.
- <http://en.wikipedia.org/wiki/ROT13>: the Wikipedia ROT-13 page. This page explains ROT-13 and related schemes in some detail, and provides some examples.
- <http://www.thegeekstuff.com/2012/07/cryptography-basics/>: Introduction to Cryptography Basic Principles. This page from The Geek Stuff website gives an overview of some of the techniques and uses of cryptography.
- <http://www.oit.umn.edu/safe-computing>: the U of M Safe Computing website.
- <http://www.youtube.com/watch?v=Wc1d0w4j3J8>: Explaining Computer Security. A YouTube video containing practical, general advice about personal computer security.

- <http://www.dhs.gov/topic/cybersecurity>: The U.S. Department of Homeland Security cybersecurity page. This page contains a wealth of information on cybersecurity and related topics.
- <http://www.dhs.gov/critical-infrastructure-sectors>: The U.S. Department of Homeland Security critical infrastructure page. This page lists a number of sectors including not only information technology, but also others such as communications, energy, and financial services.
- <http://www.schneier.com/>: Bruce Schneier Website. Schneier is an expert on security and related topics such as privacy. He writes both advanced works for experts as well as more accessible articles for popular audiences. The material here isn't necessarily closely related to what we covered in class, and you might not agree with everything Schneier writes, but this site provides interesting additional material for anyone interested in learning more about this area.



# Chapter 8

## Computer Science, Numbers, and Counting

In computer science numbers matter, including very small and very large ones.

### 8.1 Introduction

#### 8.1.1 Introductory Problem

A large part of computer graphics is modeling, for instance modeling a character in a computer-generated film, modeling a scene in an architectural simulation, or modeling a mechanical part of an automobile. One commonly-used technique models the surface of a solid object as a triangular mesh. That is, the object surface is represented as a large number of connected triangles.<sup>1</sup>

Graphics scenes can be complicated. They might contain many objects. Moreover, the triangular mesh for each object might contain many triangles in order to represent the surface reasonably accurately. Therefore, the size of the files containing scenes can be a concern. Consider the following problem.<sup>2</sup>

Suppose you are composing a computer graphics scene consisting of 100 objects. Each object is modeled as a triangular mesh consisting of an average of 100 triangles. Each triangle has three vertices, each of which has an  $x$ ,  $y$ , and  $z$  coordinate. And each coordinate is stored as a 32-bit number. Roughly how much space would it take to store this scene if you stored every coordinate of every triangle?

Computer practitioners often find themselves doing problems like this. Here are additional examples:

---

<sup>1</sup>Sometimes quadrilaterals (four-sided shapes) or more generally polygons (many-sided shapes) are used instead of, or in addition to, triangles. The mesh is then called a *quadrilateral* or *polygonal mesh*.

<sup>2</sup>A solution is in Section 8.4 below.

1. Suppose you have a 4GB file. How long will it take to download if your average download rate is 1Mbps?
2. Suppose a computer system allows passwords that are (exactly) eight characters long, with each character being an upper-case or lower-case alphabetic character or a digit 0 – 9. How many possible passwords are there? And if a malicious hacker writes a program that can check one thousand passwords per second, what is the chance that he or she will be able to access your account within five minutes?
3. Suppose you are managing a programming project. You must choose a three-person team from a group of five people. How many possible teams are there?

### 8.1.2 Overview

**Question 1:** What comes to mind when you think about “numbers and computer science?” Write a paragraph, make a list, sketch a diagram, etc.

What comes to mind? One common image is the computer as a “big calculator.” Viewing computers as massive calculators is one historical perspective on computing, as the following examples show:

- The term “computer” used to refer to humans who did calculations.
- One group of “ancestors” of computers were business machines made by companies such as Burroughs and IBM, that often did numerical business computations.<sup>3</sup>
- Some early pre-computers and early computers were built to do extensive numerical computations, such as ballistics computations during WWII.
- In general, both historically and currently, there are many important scientific, engineering and other applications that rely on computers’ abilities to do many, many arithmetic operations very rapidly.

However, there are also many other connections between computer science and numbers. One area that we’ll examine in some detail in this chapter is *counting* problems: what they are, why they are important, what some examples are, what background information is useful to know about them, and what tips or techniques are useful in solving them. The main theme of this chapter is that in computer science numbers matter, including very large and very small ones.

---

<sup>3</sup>Photographs of such computing machines are available on the web, for example see the photograph archive at the Charles Babbage Institute <http://www.cbi.umn.edu>

### 8.1.3 What Are Counting Problems and Why Are They Important?

Counting problems consist of arithmetic computations to answer fundamental problems such as “how many?” or “how long?” How large is a certain file likely to be? How many servers does a company need to meet peak customer demand? How long will it take for a person to download a file? How many different files names are there of a certain type?

**Question 2:** Give an example of a computer science counting problem that you can think of, and that is different from those mentioned above.

**Question 3:** Give an example of a counting problem that you can think of from an area other than computer science.

Counting problems arise frequently in many areas of computer science. They are important not only to individuals who might want to figure out items such as how many song files can fit on their computer’s hard disk, or how long it might take to download a large file over a slow connection, but also to computer practitioners and researchers, to businesses, and to government organizations.

Counting problems are a sufficiently important topic that they are usually part of a computer science curriculum. For example, at the University of Minnesota-Twin Cities, counting problems are a prominent topic in CSci 2011, Discrete Structures of Computer Science, which is a required class for computer science and computer engineering majors.

Counting problems are usually not deep problems: to solve them often involves just the usual arithmetic operations of multiplication, division, addition, subtraction, and exponentiation. However, there are a few challenges associated with the problems. Sometimes — as in the graphics problem beginning this chapter — there are many quantities involved; sometimes it is not always obvious how to combine the different quantities; sometimes there are extra conversions (for example, memory sizes are usually given in *bytes* but download rates in *bits* per second); sometimes the problems involve very large or very small numbers; and sometimes problems use unfamiliar terminology such as “petabytes” or “nanoseconds.” Moreover, there are a variety of different types of counting problems. So there is not a “one size fits all” formula or technique to apply in all situations.

### 8.1.4 The Liberal Education Requirements

How do counting problems relate to the liberal education themes of CSci 1001? The link to the mathematical thinking core theme is obvious — counting and arithmetic is one branch of mathematics.<sup>4</sup>

How, if at all, is counting related to the technology and society theme?

Consider the general problem of downloading files. Customers who wish to download content online are willing to wait a certain amount of time, but not too long. Nowadays we take services such as iTunes and Netflix for granted, but it was not always the case

---

<sup>4</sup>In fact, when people think of mathematics they often think of arithmetic. However, just as computer science is much broader than computer programming, mathematics is much broader than arithmetic.

that downloading a song file or downloading a movie was feasible. Companies such as Apple and Netflix needed to do a number of calculations involving download times to guarantee that their services would have a large enough set of potential customers.

Consider buying an audio file of a song over the Internet from iTunes or from a similar service. A typical song file is about 4MB in size (the size will of course vary depending, for example, on the song duration). Back when the most common connection was a 56Kbps (kilobits per second) download connection, the download time would be

$$\frac{(4 \text{ megabytes})(\text{about } 1000000 \text{ bytes/megabyte})(8 \text{ bits/byte})}{(56 \text{ kilobits/sec})(\text{about } 1000 \text{ bits/kilobit})} \approx 570 \text{ sec}$$

or about nine and a half minutes.<sup>5</sup> While this is not an outrageous amount of time, not many people are willing to wait that long for one song file.

However, Internet connection speeds have been continually increasing. If a person has a 1 Mbps (megabit per second) average download rate, then the download time decreases to approximately 32 seconds, a much more acceptable download time.

When Apple was setting up iTunes, the company needed to do a number of download computations on how long it would take potential customers with various download rates to download a typical song or a typical album. Apple calculated that the durations would provide enough potential customers that iTunes would be feasible in terms of acceptable download times.

This business planning problem is just one example of how counting problems relate to the technology and society theme. More generally, counting problems can be important to individuals, businesses, and governments because they answer questions of feasibility, planning, likelihood of occurrence, need for resources, etc.

### 8.1.5 Goals

Here is what you should be able to do once we finish this topic:

1. Given a counting problem, be able to solve it, and explain how you obtained your answer.
2. Be able to use counting problems to answer questions related to what computer capabilities are feasible now or in the future.
3. Know and be able to use terminology such as “gigabytes”, “nanoseconds”, etc. that often arise in computer-related counting problems.

---

<sup>5</sup>Notice that getting an *exact* answer for a problem of this type is neither important nor even possible: the 4MB file size is just a rough size since typical song files are rarely exactly 4MB in size. And there are a number of factors that affect actual download rates as well. More generally, while there are some types of counting problems where an exact answer is important, in many we will just look for a *reasonable rough approximation*.

## 8.2 Solving Counting Problems

There are a variety of techniques for solving counting problems. In this section we'll look at further examples of counting problems, as well as tips and techniques for solving them.

### 8.2.1 Some General Tips

Consider the following counting problems, some of which are similar to examples you've seen before, and some of which are not:

1. How long does it take to download a 3.6GB file over a 500Kbps network connection?
2. How much storage (in KB or MB) does it take to store a 1000 x 1000 pixel image, where each pixel uses 24 bit color, and the image is compressed 30:1?
3. How many possible passwords are there where the password must be 8 or 9 characters long, start with a lower case alphabetic character, and have as remaining characters any of 94 characters that consist of lower case alphabetic characters, upper case alphabetic characters, digits 0 – 9, or any of 32 punctuation characters?

Counting problems usually do not involve very complicated mathematics, but can nonetheless be challenging. For example, they often involve many quantities and computations, might involve very large or very small numbers, and might require you to know computer terminology. This section contains some hints on doing these types of problems:

- *Read the problem carefully.* Make sure you understand what the problem is asking, and understand all the items in the problem statement.
- *Combine quantities correctly.* For example, in the first problem above, you are given a file size and a download speed. To get the desired outcome, a duration, you need to divide the file size by the download speed.
- *Keep track of units.* This is not only good general strategy, but also helps ensure that you don't miss any conversions not explicitly stated in the problem (such as bits to bytes, or bytes to megabytes). It also furnishes a partial check on your answer (e.g., if you get an answer of 2KB for a download time, something is wrong since the answer should be an amount of time, not a memory size).

Be particularly careful of bits and bytes. File sizes are usually given in term of bytes, while transmission speeds are usually given in terms of bits per second. For example, in the first problem above, the file size is in gigabytes, and the download speed is in kilobits per second. So the answer will require converting bits to bytes, or bytes to bits.

- *Be familiar with the prefixes such as mega-, giga-, etc..* Many computer counting problems involve gigabytes, kilobits, nanoseconds, etc. Remember, for instance that a gigabyte is approximately a billion bytes. And a kilobit is approximately a thousand bits.
- *Be familiar with exponentiation.* Computer science counting problems usually use extensive addition, multiplication, subtraction, and/or division, which you should be comfortable with. However, often the problems will also use exponentiation. If you haven't used exponentiation recently, find an online tutorial (such as the one at listed at the end of this chapter) and review the rules on how it works.
- *Be familiar with scientific notation.* Computer science problems often involve very large numbers or very small numbers. These are often represented in scientific representation format. For example the number  $1.06 \times 10^6$ , which has a *mantissa* 1.06, and an exponent 6, is an efficient way of representing 1,060,000. Or  $-1.2 \times 10^{-6}$  is a way of representing  $-.0000012$ . If you are not familiar with scientific notation, learn about it using an online resource such as the one listed at the end of this chapter.
- *Know when to do which types of operations (e.g., add vs. multiply).* Note in the third problem above you need to *add* together the number of passwords of each different possible length. However, the first two problems just require multiplication and/or division.
- *Be familiar with different types of counting problems and the principles involved to solve them.* Some counting problems involve powers of two. Some involve ordered elements. Some involve unordered collections. Some allow items to be repeated. Some do not. Be familiar enough with the different types of problems that you know which counting technique or techniques to apply to solve a problem.

## 8.2.2 Example of Solving a Problem

Let's solve the third problem above keeping these tips in mind. Here is the problem statement again:

How many possible passwords are there where the password must be 8 or 9 characters long, start with a lower case alphabetic character, and have as remaining characters any of 94 characters that consist of lower case alphabetic characters, upper case alphabetic characters, digits 0 – 9, or any of 32 punctuation characters?

Make sure you understand what the problem is asking. Think about what type of problem this is, and what type of operations you will need to use to solve it. Note that you can break the problem up into two subproblems — the number of 8 character passwords and the number of 9 character passwords. Does this help? Take a moment and describe what steps you would do to solve this problem before reading further.

Breaking the count into an 8-character count and a 9-character count is useful. Specifically, if you can calculate these two quantities, then we add them to get a solution. So first let's consider the number of passwords that are 8 characters long. One way to think of this problem is as a "fill in the slots" type of problem: there are 8 slots; the first must be filled with any of the 26 lower case characters, and each remaining slot can be filled with any of 94 characters. So the answer is  $26 \times 94 \times 94 \times \cdots \times 94 = 26 \times 94^7$ .

Similarly, the number of passwords that are 9 characters long is  $26 \times 94^8$ . So the overall answer is  $26 \times (94^7 + 94^8)$ . This is a large number, about  $1.6 \times 10^{17}$ .

Before going on, let's consider the size of that number. Recall a billion is  $10^9$ , a trillion is  $10^{12}$ , and a quadrillion is  $10^{15}$ . So the number of passwords in this problem is about 160 quadrillion. This is good in this context — you want there to be a large number of passwords so that a malicious hacker trying random passwords or a "brute force" attack cannot discover your (or anyone else's) password.

There are other examples where the large numbers that result from counting problems are also useful. For example, how many different possible images are there are a certain size? We don't want to "run out" of distinct icons to represent items, for instance. Or how many different ways are there to combine musical notes to get a melody? Again, we don't want to "run out."

But there are also examples where the large number of possibilities is problematic. If we need to do a task that processes every possibility, then we might need to perform a very, very large number of calculations — perhaps so many that the task is not feasible, at least not using a brute force approach. For instance, suppose we are trying to decrypt messages as part of a criminal investigation. Using a brute force approach on any minimally sophisticated type of encryption will not work in any reasonable amount of time.

Next, let's return to the problem and look at two important variants:

- a. Suppose (i) there is no restriction on the first character, and (ii) that characters cannot be repeated in the password.
- b. Suppose you want to count the number of ways to choose any 8 characters out of the possible 94. That is, (i) there is no restriction on the first character; (ii) characters cannot be repeated; (iii) order does not matter, for example 'ABCDEFGH' is the same as 'HGFEDCBA'.

These two variants illustrate common types of counting problems. Specifically, sometimes items can be repeated and sometimes they cannot. And sometimes order matters and sometimes it does not. So two questions to ask yourself when solving counting problems are "is repetition allowed?" and "does order matter?"

To solve part (a), note it is still a "fill in the slots" problem. There are 8 slots, and 94 possible choices for the first position. Once this position is filled, there are 93 remaining choices for the second position, then 92 for the third, and so on. So the solution is  $94 \times 93 \times 92 \times \cdots \times 87$ , or around  $4.5 \times 10^{15}$ .

Part (b) is a *combination* problem. The number of ways of choosing  $k$  distinct items from a group of  $n$  distinct items, when order does not matter, is  $n!/(k!(n-k)!)$ . (Recall  $n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$ .) Some books use the shortened notation  $C(n, k)$  or  $\binom{n}{k}$  for this number. For this particular problem we are choosing 8 of 94 items, so the answer is  $94!/((8!)(86!))$ , which is about  $1.1 \times 10^{11}$ .

### 8.2.3 Important Quantities

In previous chapters, such as in the data representation and the machine organization chapters, you saw some important terminology related to amounts of memory and to fractions of a second. For example, a GFLOP or gigaflop is  $2^{30}$ , or roughly one billion, floating point operations per second. Here are two tables you have seen previously. Figure 8.1 lists important amounts of memory, and Figure 8.2 lists important fractions of a second.

Figure 8.1: Some Important Memory Amounts

Amount	Power of Two	Approximation
1 kilobyte (KB)	$2^{10}$ bytes	thousand bytes
1 megabyte (MB)	$2^{20}$ bytes	million bytes
1 gigabyte (GB)	$2^{30}$ bytes	billion bytes
1 terabyte (TB)	$2^{40}$ bytes	trillion bytes
1 petabyte (PB)	$2^{50}$ bytes	quadrillion bytes
1 exabyte (EB)	$2^{60}$ bytes	quintillion bytes

Figure 8.2: Some Important Fractional Time Durations

Name	Duration	Duration (in words)
1 millisecond (ms)	$10^{-3}$ seconds	one-thousandth of a second
1 microsecond ( $\mu$ s)	$10^{-6}$ seconds	one-millionth of a second
1 nanosecond (ns)	$10^{-9}$ seconds	one-billionth of a second
1 picosecond (ps)	$10^{-12}$ seconds	one-trillionth of a second
1 femtosecond (fs)	$10^{-15}$ seconds	one-quadrillionth second

There are still larger or smaller quantities of these types. For example, you can search online for names of even larger memory sizes.

### 8.2.4 A Useful Formula

Suppose you are organizing a sports league that will have “round robin” play, that is, every team will play every other team once. How many total games will there be if there are 10 teams in the league?

One way to solve this problem is to note the first team must play each of the other nine teams. The second team similarly also has to play each of the other nine teams, but you’ve already counted the game between the first team and second team. So there are eight additional games involving the second team. Similarly there are seven additional games involving the third team, and so on. The answer is therefore

$$9 + 8 + 7 + \cdots + 2 + 1 = 45.$$

Next suppose that the following season the league expands to 12 teams. Now the number of games becomes

$$11 + 10 + 9 + \cdots + 2 + 1 = 66.$$

Then suppose it expands to 16 teams, then to 18, etc. It would be useful to have a general formula rather than having to add many numbers each time. In fact, there is such a formula. Suppose you have  $n$  items, and want to count the number of different sets of two items. The formula is

$$\sum_{i=1}^{n-1} i = (n-1)n/2.$$

The notation on the left means we are taking the sum of all  $i$ -values as  $i$  ranges between 1 and  $n-1$ . So the formula states that the sum of the numbers between 1 and  $n-1$ , inclusive, equals  $(n-1)n/2$ . In the original example above  $n = 10$ , and the formula yields  $(9 \times 10)/2 = 45$ , which is the same number as obtained above. When  $n = 12$  the formula yields  $(11 \times 12)/2 = 66$ , which again agrees with what we obtained above. And when  $n = 16$  and then 18, we get  $(15 \times 16)/2 = 120$  and  $(17 \times 18)/2 = 153$ , respectively.

This is a useful formula to remember since it comes up often in computer science and elsewhere: suppose you have  $n$  computers, each of which is connected to each other computer. How many connections are there? Suppose you have to check each item in a list with each other item in a list to see if there are any duplicates. How many item-item checks will you need to do? Suppose you want each person at a party to talk for at least a minute with each other person. How many such conversations are there?<sup>6</sup>

---

<sup>6</sup>One caution: note the sum is between 1 and  $n-1$ . If you are summing the first  $n$  integers — for example, if you are counting connections between computers, but a computer can also be connected to itself — then you need to use the related formula

$$\sum_{i=1}^n i = n(n+1)/2.$$

### 8.3 Example Problems

This section contains example problems.

**Problem 1:** How many filenames are there that are six characters long, where the first character can be any of the 52 upper or lower case letters, and the remaining five characters can be any upper case letter, lower case letter, or digit 0 – 9?

**Problem 2:** Suppose you have a  $1000 \times 1000$  image, with each pixel in the image represented as a 24-bit color. The image is compressed 30:1.

- (a) Approximately how much memory does it take to store this image?
- (b) How long would it take to transmit this image over a 1Mbps channel?
- (c) Suppose you had an algorithm that analyzed the image, using 9 floating point operations for each pixel (in the uncompressed image). Suppose further that you are working on a 2 GFLOPS computer. Approximately how long will it take the computer to do these operations?

**Problem 3:** For each part below, set up the calculations, and then find the answer as a specific number. You may give large numbers as approximations rather than as their exact value. Additionally, classify the size of each numeric answer according to the following:

- “some”: numbers less than one hundred.
- “large”: numbers greater than or equal to one hundred, but less than a million.
- “colossal”: numbers greater than or equal to a million, but less than a billion.
- “mammoth”: numbers greater than or equal to a billion, but less than a trillion.
- “gargantuan”: numbers greater than or equal to a trillion.

Suppose you are investigating a computer crime. You have partial information, but are working on establishing a communication timeline for the suspect. You know the suspect sent out 12 email messages during the time period you are investigating. Each email message went to a single person (so no cc’ing, etc.). However, because the suspect and his associates used anonymity tools you do not know which email messages were sent to whom when. Answer the following questions. Remember to classify your answers as mentioned above.

- (a) Suppose the suspect had 20 known associates, and each email message went to one of these associates. How many ways are there to select 12 associates out of this group of 20 if order matters and no associate received more than one email message?
- (b) How many ways are there in problem (a) if order does *not* matter and no associate received more than one email message?

(c) How many ways are there in problem (a) if order matters, but associates might have received more than one of the 12 email messages? (For example, it's possible all 12 went to a single associate.)

(d) Suppose that due to new evidence you limit the suspect's list of associates to 15 people. Now how many possibilities are there if order matters and each email message went to a different person?

(e) Suppose you get still further new evidence: the first email went to associate X or associate Y. The next 4 emails went to people from a group of 5 associates, but some people in this group might have received more than one email message. The last 7 emails went to 7 *distinct* people from a group of 10 associates. Assume order matters. How many different possibilities are there?

## 8.4 Problem Solutions

**Solution to Introductory Problem:** Since there are 100 objects, each containing an average of 100 triangles, each of which has 3 vertices, each of which has 3 coordinates, each of which takes 32 bits (which equals 4 bytes) to store, we get a total of

$$100 \text{ objects} \times 100 \frac{\text{triangles}}{\text{object}} \times 3 \frac{\text{vertices}}{\text{triangle}} \times 3 \frac{\text{coordinates}}{\text{vertex}} \times 4 \frac{\text{bytes}}{\text{coordinate}}.$$

This is 360,000 bytes, or approximately 360KB.

**Solution to Problem 1:** Think of this as filling in six slots. There are 52 possibilities for the first slot. Then there are  $26 + 26 + 10 = 62$  possibilities for the second, another 62 possibilities for the third, another 62 for the fourth, etc. So there are  $52 \times 62 \times 62 \times 62 \times 62 \times 62$  possibilities total. This equals 47,638,907,264, or about 47 billion.

**Solution to Problem 2:**

(a) The image size is

$$(1000 \times 1000) \text{ pixels} \times 24 \text{ bits/pixel} \times (1/30)/(8 \text{ bits/byte}) = 100,000 \text{ bytes}$$

or about 100KB.

(b) The answer is  $100,000 \text{ bytes} \times (8 \text{ bits/byte})/1,000,000 \text{ bps} \approx .8 \text{ seconds}$ . (Don't forget the bytes to bits conversion.)

(c) The image contains  $1000 \times 1000 = 1 \text{ million pixels}$ . So the answer is  $(1 \text{ million pixels} \times 9 \text{ floating point operations/pixel})/2 \text{ billion floating point operations per second}$ . This simplifies to  $9/2000 = .0045 \text{ seconds}$ , or 4.5 milliseconds.

**Solution to Problem 3:**

- (a)  $20 \times 19 \times 18 \times \dots \times 9$  or about 6 trillion ( $6 \times 10^{12}$ ). This is a gargantuan number.
- (b) This is a combination problem. The answer is  $C(20, 12) = 20!/(12!8!) = 125,970$ . This is a large number.
- (c) This is  $20^{12}$ , which is about 4 quadrillion ( $4 \times 10^{15}$ ). This is a gargantuan number.
- (d)  $15 \times 14 \times 13 \times \dots \times 4$ , or about 220 billion ( $2.2 \times 10^{11}$ ). This is a mammoth number.
- (e) There are 2 possibilities for the first email,  $5^4$  for the next four, and  $10 \times 9 \times 8 \times \dots \times 4$  for the last seven. Multiply all these together to get an answer of 756 million ( $7.56 \times 10^8$ ). This is a colossal number.

## 8.5 Additional Questions to Think About

Here are questions to think about. We will explore some of these further in class.

1. What does this topic tell us about the nature of computer science, and about what computer scientists and computer practitioners do?
2. What does this topic tell us about mathematics: what does it show about how mathematics is used in computer science, and what does it illustrate about mathematics itself?
3. What does this topic tell us about society and technology: what does it tell us about both computers and society, as well as about technology and society in general?
4. Subsection 8.1.4 gave one example of how counting problems relate to the technology and society theme in this class: that they are often used by businesses to figure out what is (or soon will be) feasible with computers. What other ways can you think of that counting problems relate to the technology and society theme?

### 8.5.1 Some Further, Related Problems

Let's use the introductory storage size problem to explore other related problems. You are invited to solve these problems on your own. No solutions are given here, so work carefully and think about how to check if your answer is likely to be correct.

1. Consider the following alternative storage scheme: suppose each object possessed about 150 distinct triangle vertices. (Note multiple triangles can share a vertex, so the number of vertices is often significantly less than three times the number of triangles.) The scheme consists of two tables. The first table has a row for each vertex, with the row having three columns: one for each coordinate. The second table has a row for each triangle, with each row having three columns which hold the indices of the triangle's vertices in the first table. An example mesh is shown in Figure 8.3, with the first table shown in Figure 8.4 and the second in Figure 8.5.

Figure 8.3: A simple triangular mesh

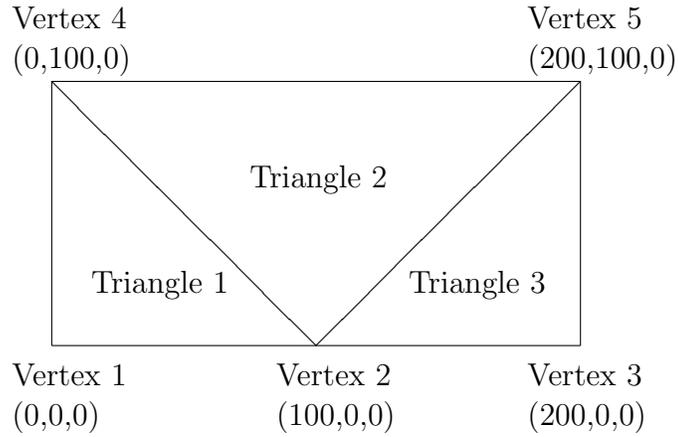


Figure 8.4: The Vertex Coordinate Table

Vertex Number	$x$	$y$	$z$
1	0.0	0.0	0.0
2	100.0	0.0	0.0
3	200.0	0.0	0.0
4	0.0	100.0	0.0
5	200.0	100.0	0.0

Figure 8.5: The Triangle and Vertex Number Table

Triangle Number	First Vertex Index	Second Vertex Index	Third Vertex Index
1	1	2	4
2	2	5	4
3	2	3	5

- (a) How much less space does this file take than the original storage scheme?
  - (b) How much space this alternative scheme saves is dependent on the quantities in the problem, for example, the number of triangles and number of distinct vertices. When is this alternative scheme likely to result in significant savings? When is it likely to result in only small savings?
2. What other reasons — beyond storage space — might make one storage scheme preferable to the other?<sup>7</sup>

## 8.6 Additional Resources

Here are a few additional resources that might be useful:

- <http://www.mathsisfun.com/index-notation-powers.html> — *Math is Fun* page on scientific notation.
- <http://www.mathsisfun.com/algebra/exponent-laws.html> — *Math is Fun* page on laws of exponentiation.

---

<sup>7</sup>For a complicated problem there will be often be different ways to structure and store the data, and each will have advantages and disadvantages. So computer practitioners will need to weigh the trade-offs of each approach when deciding which to use.

# Chapter 9

## Algorithmic Complexity

Does it scale?

### 9.1 Introduction

#### 9.1.1 Introductory Problem

Suppose you have a computer animation involving 100 different objects. As the animation progresses the objects move, and for each frame in the animation the computer needs to detect if any objects have collided. Assume further you have a check that computes if two given objects are colliding at the present time.<sup>1</sup>

Questions:<sup>2</sup>

1. How many times will the computer need to perform this check for a frame in the animation if you want to check each pair of objects?
2. Suppose that you want a more complicated animation, so you create additional objects and have 200 objects in your animation. Now how many times does the computer need to run the check?
3. What is the ratio of the number of times in Question 1 to that in Question 2? For example, does the number of times double?
4. In general, suppose you have  $n$  objects in the scene. (a) What is the number of times you'll need to run the collision detection check? (b) How does this increase as  $n$  increases? For example, if  $n$  doubles, does the number of times double?

---

<sup>1</sup>Whether any two given objects are colliding is itself a very interesting and complicated problem. For example, if the objects are modeled as collections of triangles, then a straightforward approach is to intersect each triangle from the first object with each from the second. This can get complicated very quickly, and so computer animations often use a variety of clever techniques to make collision detection algorithms more efficient.

<sup>2</sup>Solutions are in Section 9.8.1 below.

## 9.1.2 Overview and Motivation

There is more data being acquired, stored, and analyzed than ever before in human history. Astronomy data collected by telescopes, space probes, etc. tells us about the nature of the universe we inhabit. Government census data tells the government about its populace's characteristics: how many people live in the various states, counties, and cities? What are their occupations? What do age demographics look like? Business data tells companies who buys their products, whether there are seasonal fluctuation in certain product purchases, etc.

One key question in analyzing large data sets is “how long will the operations take if the data set size increases?” For example, suppose a company has a boom year, and its sales and number of customers both increase significantly, meaning their databases all become twice as large. Will the company be able to handle these larger databases with their current resources — their database programs, number of computers, network connections, etc.? And will the various database operations that the company does still be doable in a reasonable amount of time?

The question “Does it scale?” is therefore an important one. It is important in other areas of science and engineering (for example, chemical engineers take chemical reactions that can be done in a laboratory, and try to turn them into feasible and efficient industrial-scale processes). And it is important to computer scientists when they analyze algorithms. A computational process that can be done for a small problem might or might not be feasible for a large problem.

This chapter therefore explores *algorithmic* or *computational complexity*: how does the number of operations an algorithm must perform grow as the problem size increases? Specifically, this chapter explains how computer scientists measure algorithmic complexity, looks further at why this is important, and provides examples of some of the types of problems you should be able to do related to this topic.

## 9.1.3 Further Explanation of Algorithmic Complexity

Algorithmic complexity is one way, indeed the primary way, that computer scientists measure algorithmic efficiency.<sup>3</sup> Algorithmic complexity is not an exact measure of how long an algorithm will take to run. In fact, there are many factors that make an exact running time difficult or impossible to measure. For example, the duration an algorithm takes to solve a problem is affected by factors such as

- how fast the computer running the program is;
- how efficient the computer language used to implement the algorithm is;

---

<sup>3</sup>By algorithmic or computational complexity we mean *time* complexity. There is an associated concept of *space complexity*, or how the amount of computer memory an algorithm uses grows as the problem size increases. Space complexity can be important, especially for algorithms working with very large amounts of data. However, computer scientists are usually more concerned with time complexity, and so we will restrict our attention to time complexity in this class.

- whether the computer is doing any other tasks at the same time it is running the algorithm (e.g., is it also streaming music?);
- the size of the input to the algorithm.

Occasionally computer scientists or engineers will do “benchmark tests” that check the raw time an implementation of an algorithm takes on specific test input under controlled conditions. However, more often they measure algorithm efficiency theoretically by studying how the running time of the algorithm increases as the problem size increases. For example, if the size of an input data set doubles, will the approximate time an algorithm takes also double? Or will it increase by a factor of four? Or maybe eight? Or maybe more?

There is a mathematical definition of algorithmic complexity. You are welcome to look it up if you are curious. Basically what it says is that for a sufficiently large problem size we can get an upper bound (or “big-O” complexity (pronounced “big-Oh”)) or an upper *and* a lower bound (or “big- $\Theta$ ” (pronounced “big Theta”)) on the growth of the number of operations the algorithm takes. However, for the purposes of this class an intuitive understanding of what algorithmic complexity is and is not is sufficient.

Before looking at an intuitive description of complexity, let’s consider three other fundamental questions: Why are computer scientists interested in growth for large problem sizes? What do we mean by a problem’s “size?” And what does “sufficiently large” mean anyway — is it 100? 1,000? 1,000,000?

Computer scientists are interested in growth for large problem sizes for a few reasons, including:

- Computers and algorithms are sufficiently fast that most small problems can be solved very quickly, and so the running time for small problems often doesn’t matter.
- For some problems, the amount of work done required increases dramatically as the problem size increases. So even if these problems can be solved quickly for small problem sizes, they might not be solvable in a reasonable amount of time for larger problem sizes.
- There are some truly large problems, and the amount of data scientists and others are gathering for these problems is increasing. Whether algorithms can handle these large and growing data sets in a reasonable amount of time depends on the algorithm’s complexity.
- Many cutting-edge applications nowadays work with large problem sizes.

“Problem size” usually means the number of items in an algorithm’s input. If you are measuring a sorting algorithm, the problem size is usually the number of items to be sorted.<sup>4</sup> If you are calculating pay amounts and printing paychecks for all your employees,

---

<sup>4</sup>The word “usually” is used here because sometimes the problem size depends on other characteristics of the data. For example, some sorting algorithms also depend on the maximum number of digits or characters in the items to sort.

it is usually the number of employees. If you are checking collisions between each pair of objects in a graphics scene, it is usually the number of objects.

What constitutes “sufficiently large” varies from problem to problem. However, this turns out not to be a major concern for some of the same reasons as were just mentioned. For example, as more and more data is gathered problem sizes grow, and so if a data set is not “sufficiently large” now, it might be soon.

Intuitively, computational or algorithmic complexity measures how the number of operations an algorithm performs grows as the problem size grows. And since the time an algorithm takes is closely related to the number of operations the algorithm performs, it is also a measure of how the time increases as the problem size does. For example, if an algorithm has “linear” complexity (see below) then if the problem size doubles, the amount of time approximately doubles; if the problem size triples, the amount of time approximately triples; if the problem size increases by a factor of 10, then the amount of time increases by about a factor of 10.

In addition to knowing what algorithmic complexity is, it is important to remember what it is not. It is not

- an exact count of the running time or number of operations of an algorithm;
- a measure of how the algorithm performs for small problem sizes;
- a means of comparing two algorithms’ running times on small data sets.

We’ll explore examples of computing the algorithmic complexity of specific algorithms after a few more preliminaries.

### 9.1.4 Topic Goals

After completing this section you should be able to do the following:

1. Be able to explain what algorithmic complexity is, and why it is an important topic in computer science.
2. Be able to explain how computer scientists measure algorithmic complexity. And be able to explain what these techniques do and do not measure.
3. Given a description of an algorithm, be able to calculate its complexity.

### 9.1.5 Algorithmic Complexity and Mathematics

Algorithmic complexity is related to counting problems. Calculating the complexity often requires counting operations. However, unlike many counting problems where the answer is a specific number, in complexity the answer is a function. In particular, the answer is a function of the problem size. Once you have found this function, algorithmic complexity

asks how quickly the functions grows. So algorithmic complexity, at least in its fundamental form, involves two subareas of mathematics: counting problems and analyzing function behavior.

### 9.1.6 Algorithmic Complexity and Technology and Society

As mentioned above, humans are gathering more data than ever before in human history. What we can do with these increasing amounts of data is dependent on how algorithms scale. If you increase a dataset size by a factor of 10 and a computational task goes from taking one hour to ten hours, then it is probably still doable — you just need to wait longer for the result (or get a faster computer, etc.). If the task goes from taking one hour to one thousand hours, then you might or might not have the time and other resources needed. If a task goes from one hour to one trillion hours, then you probably will not get an answer in your lifetime.

Many of the most interesting problems in computer science, in science and engineering, in biology and medicine, and elsewhere involve large amounts of data. Organizations that are dealing with these issues are not only bodies such as NASA working with space exploration data, but also many companies, government bodies, and other organizations working with business, social, or other data. For example, two computer-related companies for whom issues of scale are a constant concern are Google and Facebook, both of which store, process, and transmit massive amount of data for truly large numbers of users.

## 9.2 How To Measure Algorithmic Complexity?

### 9.2.1 Big- $O$ and Big- $\Theta$

Computer scientists often use big- $O$  or big- $\Theta$  notation to indicate an algorithm's complexity. Big- $O$  provides an upper bound on growth. If an algorithm is  $O(n^2)$  (pronounced “big Oh of  $n$  squared”), the algorithm's time will, when a sufficiently large problem size is doubled, take at most about four times as long. Note this provides an upper bound — that is, an *at most* bound. It's possible, for example, that the algorithm might only take twice as long.

Big- $\Theta$  is more precise, and provides both an upper and a lower bound. If a sufficiently large problem size is doubled, a  $\Theta(n^2)$  algorithm will take about four times as long. This is pronounced as “big Theta of  $n$  squared” and is also called “order  $n$  squared” or *quadratic* complexity.

To complicate matters further, computer scientists sometimes distinguish between *best case*, *average case*, and *worst case* complexity. Suppose you are searching for a specific employee address in a list of  $n$  addresses. It might be the first address, in which case you're done once you checked the first item. So in this case you checked a constant number of items, and the best case complexity is  $\Theta(1)$ . It might be the last item, in which case

you've checked all  $n$  items. So the worst case complexity is  $\Theta(n)$ . The average complexity is more difficult. Let's assume the desired address is equally likely to be at any position in the list. So the chance of it being in any particular place is  $1/n$ . The number of items checked will be 1 if the item is in the first location, or 2 if it is in the second location, and so on. So the average number of comparisons is  $(1 \times 1/n) + (2 \times 1/n) + \dots + (n \times 1/n)$ . This sum turns out to be  $(n + 1)/2$ . Note that although the number of checks is about half as many as in the worst case, the number of operations still grows proportional to  $n$ , and so is still  $\Theta(n)$ . (See the subsection below for why computer scientists write this as  $\Theta(n)$  rather than as  $\Theta((n + 1)/2)$ .)

For many problems we look at in this class the best case, worst case, and average case complexity will all be the same. For those problems where they are not, assume — unless otherwise specified — that we are interested in worst case complexity.

The list below explains common orders of growth. Many of the functions involved also appear in Table 9.1. The columns in that table represent different functions, the rows different problem sizes. Note the problem sizes double, from 10 to 20, then to 40, then to 80.

Figure 9.1: The growth of some functions

	1	$\lg(n)$	$n$	$n \lg(n)$	$n^2$	$n^3$	$2^n$	$n!$
$n = 10$	1	3.3	10	33	100	1000	1024	$3.6 \times 10^6$
$n = 20$	1	4.3	20	86	400	8000	$1.0 \times 10^6$	$2.4 \times 10^{18}$
$n = 40$	1	5.3	40	213	1600	64000	$1.1 \times 10^{12}$	$18.2 \times 10^{47}$
$n = 80$	1	6.3	80	506	6400	512000	$1.2 \times 10^{24}$	$7.2 \times 10^{118}$

- $\Theta(1)$  or constant complexity: the algorithm time is roughly the same regardless of problem size. This is very nice, but also very rare.
- $\Theta(\lg n)$  or logarithmic complexity: the algorithm time grows a constant amount if the problem size doubles. This is a very slow rate of growth. An example of a logarithmic complexity algorithm is *binary search*, which is an efficient searching algorithm. (See the section below on logarithms for more information on the  $\lg$  function.)
- $\Theta(n)$  or linear complexity: the algorithm time grows proportional to the growth in the problem size. For example, if the problem size doubles the algorithm takes roughly two times as long, if it triples the algorithm takes roughly three times as long. If it increases by a factor of ten the algorithm takes roughly ten times as long.
- $\Theta(n \lg n)$  complexity: the algorithm time grows slightly more than linearly. So if the problem size doubles, the algorithm takes slightly more than twice as long. An example of an order  $n \lg n$  algorithm is an efficient sort such as mergesort.

- $\Theta(n^2)$  or quadratic complexity: the algorithm time grows proportional to the square of the growth in the problem size; if the problem size doubles the algorithm takes roughly  $2^2 = 4$  times as long. If the problem size increases by a factor of 10 then the algorithm takes roughly  $10^2 = 100$  times as long. A less efficient sort such as bubble sort has quadratic complexity.
- $\Theta(n^3)$  or cubic complexity: the algorithm time grows proportional to the cube of the growth in the problem size; if the problem size doubles the algorithm takes roughly  $2^3 = 8$  times as long. If the problem size increases tenfold the algorithm takes roughly  $10^3 = 1000$  times as long.
- $\Theta(2^n)$  or exponential complexity: the algorithm time grows by a factor of roughly 2 if problem size increases by one. It is impossible to solve large exponential complexity problems exactly in a reasonable amount of time.
- $\Theta(n!)$  or factorial complexity: the algorithm time grows by a factor of  $n$  as the problem size increases from  $n - 1$  to  $n$ . An example of a factorial time complexity problem is the traveling salesman problem described below. Factorial complexity problems grow even more quickly than exponential complexity problems, and so they are also impossible to solve exactly in a reasonable amount of time.

### 9.2.2 Simplifying Big-O and Big- $\Theta$ notation

Suppose you are working for a company that is having you cross-check customer accounts. You compute the complexity of the cross-checking algorithm and find that for an input list of  $n$  accounts the algorithm performs  $n^3 + 5n + 2$  operations. Should you say that the algorithm has  $\Theta(n^3 + 5n + 2)$  time complexity?

If you said that to a computer scientist, he or she would say you were correct, but were using the terminology oddly. To see why this is so, let's examine what happens as the function  $f(n) = n^3 + 5n + 2$  grows. For example, let's compare the value of  $f(100)$  with the value of  $f(200)$ . When  $n = 100$ , the value of  $f$  is  $100^3 + 5(100) + 2 = 1,000,502$ . Notice this is not far from  $100^3 = 1,000,000$ . When  $n = 200$ , the value of  $f$  is  $200^3 + 5(200) + 2 = 8,001,002$ . Notice this is not far from  $200^3 = 8,000,000$ . Moreover, notice the ratio  $f(200)/f(100) = 7.9969\dots$  is approximately 8. That is, when the problem size doubles from 100 to 200, the number of operations increases by a factor of almost 8, which is characteristic of  $\Theta(n^3)$  complexity.

This analysis is a strong indication (although not a rigorous proof) that  $n^3 + 5n + 2$  is  $\Theta(n^3)$ . And in fact, when working with polynomials, we can drop the “lower order terms” — that is, all terms other than the one with the highest exponent. So  $n^3 + 5n + 2$  is indeed  $\Theta(n^3)$ . As another example,  $n^5 + 6n^4 - 12n^2 + 23$  is  $\Theta(n^5)$ . The reason for this simplification is that as  $n$  increases the highest order term — the term with the highest exponent — affects the function the most.

In summary, rather than saying that an algorithm that takes  $n^3 + 5n + 2$  operations has  $\Theta(n^3 + 5n + 2)$  complexity, a computer scientist would say it has  $\Theta(n^3)$  complexity. This focuses on what part of the function is dominating the growth as  $n$  gets large.

There is one more simplification when working with big-O and big- $\Theta$  notation. Consider the function  $7n^3 + 5n + 2$ . We've already seen that we wouldn't include the  $5n + 2$  when reporting the complexity. So would we say this function is  $\Theta(7n^3)$ ? Again, computer scientists would simplify this: the lead constant 7 does not affect the growth, so the complexity is  $\Theta(n^3)$ . Again,  $\Theta(7n^3)$  is not, strictly speaking, incorrect, but it doesn't focus on the fact that  $n^3$  is driving the function growth, and that the constant 7 is a "bystander."

To explore this further, evaluate  $7n^3$  at  $n = 100$  to get 7,000,000. Next, evaluate  $7n^3$  at  $n = 200$ . This gives 56,000,000, or 8 times as much. This is exactly the same ratio that we get when we take the ratio  $200^3/100^3$ . Or let's derive the ratio in another way:

$$\frac{7(200^3)}{7(100^3)} = \frac{200^3}{100^3} = \frac{8000000}{1000000} = 8.$$

In particular, notice that the 7's cancel, and so they do not affect the growth.

As a summary example, suppose we found an algorithm took  $5n^4 + 16n^3 - 2n^2 + 42$  operations. We'd simplify this by dropping the lower order terms. This would leave the highest order term,  $5n^4$ . We would then drop the constant 5, and would say the algorithm has  $\Theta(n^4)$  complexity.

### 9.3 Logarithms

Logarithms are mathematical functions that are very important in a number of fields, including computer science. One way to understand logarithms is as inverses of exponentiation. Recall that Moore's Law involves transistor density *doubling* every two years. This leads to exponential growth. To "undo" exponential growth you use logarithms.

Here's a quick example. Suppose you had a database whose size doubled every year. Suppose it started at 3MB in size, and you wanted to know how large it was after four years. Then you would double 3MB, then double the result, then double that result, and double that result. Equivalently, you could compute  $3 \times 2^4 = 3 \times 16 = 48$ MB.

Now consider a related problem. Suppose you start with a 3MB database, and know it doubles in size every year. How many years does it take until it is 48MB in size? Forget for the moment that you know, from the previous paragraph, that the answer is four years. To solve this problem you could count the number of doublings needed to get 48MB: the first doubling gives 6MB, the second 12MB, the third 24MB, and the fourth 48MB.

Equivalently, you could write the equation  $3 \times 2^t = 48$  where  $t$  is the number of doublings needed. Note this is not the type of equation we are used to solving since the unknown  $t$  is an exponent. To transform this into an easier-to-work-with form, you can

take the logarithm of both sides. You might recall there are different type of logarithms, such as common logarithms (base 10) and natural logarithms (base  $e$ , which is 2.71828...). Computer scientists most often use logarithms base 2, which is written as  $\lg$ . Specifically, the function  $\lg x$  is the number that 2 needs to be raised to in order to get  $x$ . This can be a somewhat confusing definition, but it is useful to consider logarithms of powers of 2: since  $2^0 = 1$ , we have  $\lg 1 = 0$ . Since  $2^1 = 2$ , we have  $\lg 2 = 1$ . Since  $2^2 = 4$ , we have  $\lg 4 = 2$ . And in general  $\lg 2^n = n$ .

Before we proceed further, let us examine the  $\lg$  function's characteristics. The  $\lg$  function grows very slowly: see Table 9.2 for some representative values. There are also

Figure 9.2: The  $\lg$  function

$n$	$\lg n$ (approx.)
10	3.32
100	6.64
1,000	9.97
10,000	13.28
100,000	16.61
1,000,000	19.93

some properties that are useful when manipulating logarithms:

$$\begin{aligned}\lg 2^x &= x \\ \lg a^x &= x \lg a \\ \lg(ab) &= (\lg a) + (\lg b) \\ \lg(a/b) &= (\lg a) - (\lg b)\end{aligned}$$

So let's return to the problem. We now take the equation  $3 \times 2^t = 48$ . We then take the  $\lg$  of each side, and use the properties of logarithms:

$$\begin{aligned}\lg(3 \times 2^t) &= \lg 48 \\ \lg 3 + \lg 2^t &= \lg 48 \\ \lg 2^t &= \lg 48 - \lg 3 \\ t \lg 2 &= \lg(48/3) \\ t &= \lg 16 \\ t &= \lg 2^4 \\ t &= 4.\end{aligned}$$

Of course not all problems will turn out so neatly. Sometimes you'll need to work with a quantity such as, say,  $\lg 73$ . To compute this you can use a calculator. If you can't find a

calculator with a lg function, you can use the following identity where log is the common log, i.e., log base 10:

$$\lg x = \log x / \log 2 \approx \log x / .301.$$

(The  $\approx$  sign here means “approximately equal to.”) However, another very useful skill is being able to estimate lg values without using a calculator. This is where knowing powers of 2 is useful. What power(s) of 2 is 73 close to? It is between  $2^6 = 64$  and  $2^7 = 128$ , and closer to the former than the latter. So  $\lg 73$  is slightly more than 6.

Logarithms are useful not only in some computations, but also in the rate of growth of algorithms. For example, one famous search algorithm is *binary search*, whose rate of growth is  $\Theta(\lg n)$ . And some sorting algorithms are  $\Theta(n \lg n)$ . As  $n$  gets at all large, these algorithms grow significantly more slowly than alternative  $\Theta(n)$  search and  $\Theta(n^2)$  sort algorithms. See Table 9.3, and note that the contrasts apparent there become more pronounced as  $n$  gets even larger.

Figure 9.3: Comparison of some rates of growth

$\lg n$	$n$	$n \lg n$	$n^2$
1	2	2	4
2	4	8	16
3	8	24	64
4	16	64	256
5	32	160	1024
6	64	384	4096
7	128	896	16384

## 9.4 Calculating Complexities

One useful skill for computer scientists and software developers is to be able to analyze a given algorithm’s complexity. Computing the complexity of complicated algorithms can be difficult. However, calculating the complexity of simpler algorithms is a task you should be able to do. This section contains a few simple algorithms along with derivations of their complexity.

### Example 1

```

1 Set sum = 0
2 For i = 1 to n
3   For j = 1 to n
4     Set sum = sum + a[i,j]
5 Print sum
6 Stop
```

Notice that this algorithm contains a double loop. How often is the `Set sum` operation done? Note that the  $i$  loop is executed  $n$  times, once for each of the values between 1 and  $n$ , and each time the  $i$  loop is executed the  $j$  loop is executed  $n$  times. So the total number of times is  $n \times n = n^2$ . Notice this is the most performed operation in the algorithm. Setting `sum` to 0 is done only once, as is printing the sum. (In the subsequent examples below we won't mention operations like these that are performed so few times they do not affect the final complexity.) So the dominant term in the operation count is  $n^2$ , and this is a  $\Theta(n^2)$  algorithm.

### Example 2

```

1 Set sum = 0
2 For i = 1 to n
3   For j = 1 to n
4     For k = 1 to n
5       Set sum = sum + a[i,j,k]
6 For i = 1 to n
7   For j = 1 to 10
8     Set sum = sum + b[i,j]
9 Print sum
10 Stop
```

Here the outer loop starting at line 2 is executed  $n$  times, the middle loop starting at line 3 is executed  $n$  times, and the inner loop starting at line 4 is executed  $n$  times. So this triple loop updates `sum` in line 5 a total of  $n \times n \times n$ , or  $n^3$  times. Then the outer loop starting at line 6 is executed  $n$  times and the inner loop starting at line 7 is executed 10 times, and so `sum` is updated  $n \times 10$ , or  $10n$  times in line 8. The total number of updates is therefore  $n^3 + 10n$ . Since as  $n$  gets large  $n^3$  dominates  $10n$ , this is written as  $\Theta(n^3)$ .

### Example 3

```

1 Set sum = 0
2 Set i = 1
3 While i <= n
4   Set sum = sum + i
5   Set i = i * 2
6 Print sum
7 Stop
```

This one is more difficult. Let's look at a particular value of  $n$ . Suppose  $n$  is 77. Then what values does  $i$  assume? It starts at 1, then goes to 2, then to 4, then to 8, 16, 32, 64, and finally 128. For a general  $n$ , the loop will be executed until the doubling process produces an  $i$ -value greater than  $n$ . Let  $k$  be the number of times  $i$  is doubled until it is greater than  $n$ . So  $k$  is the least integer such that  $2^k > n$ . This number is essentially  $\lg n$ : recall  $\lg$  is the logarithm base two, and  $\lg x$  is the exponent you need to raise 2 to in order to get  $x$ . So the algorithm's complexity is  $\Theta(\lg n)$ .

## 9.5 Additional Example Problems

Here are a number of additional example problems involving complexity. Solutions are in the problem answer section below.

### Example Problem 1

Suppose you are searching through an  $n$ -row,  $n$ -column,  $n$ -depth set of geological data. Give a big- $\Theta$  time complexity for each of the following algorithms. Assume `op1`, `op2`, and `op3` all take constant time.

Remember to show your work or explain your answers. Also, write your big- $\Theta$  answer as simply as possible. For example, instead of writing  $\Theta(2n^2 + 4n)$  write  $\Theta(n^2)$  since big- $\Theta$  estimates are irregardless of lower order terms (so the  $4n$  is not needed), and multiplying factors (so the 2 in  $2n^2$  is not needed).

(a)

```
1 for row = 1 to n
2   for col = 1 to n
3     for depth = 1 to n
4       do op1
5 for row = 1 to n
6   for col = 1 to n
7     do op2
8 stop
```

(b)

```
1 for row = 1 to n
2   for col = 1 to 3
3     for depth = 1 to 3
4       do op1
5 for row = 1 to n
6   for col = 1 to n
7     do op2
8 stop
```

(c)

```

1 for row = 1 to n
2   for col = 1 to 3
3     for depth = 1 to n
4       do op1
5 for row = 1 to n
6   for col = 1 to n
7     do op2
8 for row = 1 to n
9   for col = 1 to n
10    do op3
11 stop

```

(d)

```

1 for row = 1 to n
2   for col = 1 to n
3     depth = 1
4     while depth <= n
5       do op1
6       depth = depth * 2
7 for row = 1 to n
8   for col = 1 to n
9     do op2
10 stop

```

**Example Problem 2**

(a) Suppose you are doing public health research studying diabetes. You have a database consisting of 10,000 records, and have a  $\Theta(n^2)$  program (where  $n$  is the number of records) looking for correlations in the data. Suppose that when you run the program on your computer it takes about 7 minutes to finish.

(a) Suppose that you get additional records so that the database size becomes 20,000 records. Now about how long will it take to run the program?

(b) Suppose a colleague asks you to run your program for a database she has that has 100,000 records. Now about how long will it take to run the program?

(c) Suppose you have 20 databases, each of 10,000 records. You run the program on the first of these, then immediately after it is done you run it on the second, and so forth until the program has been run on all 20 databases. About how long will this take?

(d) Which takes less time: running the program on 20 databases each with 10,000 records, or running it on a single database with 100,000 records?

(e) Suppose a coworker with programming and algorithm experience claims he can improve your algorithm. You run the program on your 10,000 record database and it now takes 9

minutes. Moreover, due to a mistake the coworker made, the program now is  $\Theta(n^3)$ . Now about how long will it take if you run the algorithm on the 100,000 record database?

(f) Going back to the  $\Theta(n^2)$  algorithm that takes 7 minutes on a database of 10,000 records: suppose you increase the database size by a factor of 5, so you have a database of 50,000 records. But you also get a computer that is 5 times as fast. How long will it take your program to run using the new computer on the 50,000 record database?

### Example Problem 3:

Abrupt changes in adjacent data values often indicate something significant in the data. For example, in MRI data a large difference might indicate a transition between different tissue types, say bone and muscle. Or in a geologic data set a large difference might indicate the transition from one rock formation to another.

The following algorithm goes through an  $n$ -row,  $n$ -column,  $n$ -depth data set, and checks if there is a large difference between a data value and the next data value in the same column and at the same depth (so it is checking for differences in one direction, but not all directions). In the pseudocode below `abs` refers to the absolute value function. So, for example `abs(-5)` returns 5.

*Input:* an  $n$ -row,  $n$ -column,  $n$ -depth data set of numbers, along with a positive number *threshold*.

*Output:* a message “Large difference found” if any value differs from the next value in the same column and same depth by more than *threshold*, and a message “No large difference found” otherwise.

```

1  Get threshold, the size n, and the data set
2  Set i = 1
3  Set sigDiffFound = false
4  While i < n and sigDiffFound equals false
5      Set j = 1
6      While j <= n and sigDiffFound equals false
7          Set k = 1
8          While k <= n and sigDiffFound equals false
9              If abs(a[i,j,k] - a[i+1,j,k]) > threshold
10                 Set sigDiffFound = true
11                 Set k = k + 1
12             Set j = j + 1
13         Set i = i + 1
14 If sigDiffFound equals true
15     Print 'Large difference found'
16 Else
17     Print 'No large difference found'
18 Stop

```

Now answer the following questions. In each part do not make any assumptions on the value of  $n$ , so your answer might be a function of  $n$ . However, write your answer as a number or as a function of  $n$  and not, for example, as a big- $\Theta$  estimate.

- (a) What is the fewest number of times Line 9 is executed?
- (b) What is the largest number of times Line 9 is executed?
- (c) Suppose the number of rows, column, and depths all increased from  $n$  to  $2n$ . What is the ratio of the fewest number of times Line 9 is executed in this case to the number of times in your answer for part (a)?
- (d) Suppose the number of rows, column, and depths all increased from  $n$  to  $2n$ . What is the ratio of the largest number of times Line 9 is executed in this case to the number of times in your answer for part (b)?

## 9.6 The Traveling Salesman Problem

Are there really problems that grow extremely rapidly, so rapidly that they can't be solved in a reasonable amount of time? The answer is emphatically yes.

In some cases a problem takes a large amount of time because the underlying data set is so large. For example a wind tunnel simulation in aerospace research might involve a large number of variables at a large number of points over a large number of time steps. Or a genomic research problem might involve searching millions of DNA sequences. On the other hand, some problems are difficult because the number of computations needed grows very quickly, even if the data set involved is small. The *traveling salesman problem* is a canonical example of a problem whose complexity grows so quickly it cannot be solved for large  $n$ .

In the traveling salesman problem, a salesman needs to visit  $n$  cities, visiting each city exactly once. The salesman can choose which city to start with, then which of the other cities to visit next, then which of the remaining cities to visit after that, and so on until he or she visits all the cities. In addition to the set of cities, the problem also specifies the distance<sup>5</sup> between each pair of cities. The traveling salesman problem is to find which order of visiting the cities will result in a path of least total distance.

Because there are  $n$  choices for the starting city, then  $n - 1$  for the next city to visit, and so on, the number of possibilities is  $n \times (n - 1) \times (n - 2) \times \dots \times 3 \times 2 \times 1 = n!$ . Recall  $n!$  grows very quickly — for example  $10!$  is over 3 million. And so checking all possible paths to compute an exact solution to the traveling salesman problem is impractical when  $n$  is at all large.

The traveling salesman problem is not an isolated problem. There are many other problems that share the traveling salesman problem's rapid growth.

---

<sup>5</sup>Variants of the problem replace distance with travel time, or cost, or some other quantity. The key point is that there is some positive number associated with each pair of cities.

## 9.7 What If a Problem Takes Too Long To Solve?

Suppose you have a computational problem that takes too long for a single computer to solve. However, it is an important enough problem that you really, really would like an answer. How can you solve it? There are a variety of different techniques that might help.

One possibility is to run the problem on a faster computer, assuming you have access to one. This approach will only work to some extent. For example, if a problem scales linearly then — if all other things are equal — a computer that is twice as fast will allow you to solve a problem in half the time. However, if a problem scales exponentially then a computer that is twice as fast will not be much help.

Another possibility is to use a computer that can do many operations simultaneously. Some computers have many processors that can work in parallel. For instance, supercomputers used to perform large science and engineering simulations might have thousands of processors. Lately even personal computers have gone to multiple processors, such as “quadcore” or four-processor computers.

However, not all problems can be distributed efficiently among multiple processors. Moreover, even if a problem is amenable to a parallel approach the speed-up still might not be enough to solve the problem.

Still another approach is distributed computing. Instead of distributing a problem over different processors on a single computer, the problem is distributed over different computers. For example, a company such as Pixar that makes computer-animated films uses hundreds of computers to do the computations needed to produce a film such as *Toy Story* or *Brave*. As another example, the computations for programs such as SETI @home<sup>6</sup> (search for extraterrestrial intelligence) is distributed to many computers, including many volunteers’ personal computers. Like the previous approaches, distributed computing works for some problems, but not all.

Still another possibility is to use another algorithm. One task some computer scientists do is to devise efficient algorithms to solve new problems, or to try to devise better algorithms (or improve known algorithms) to solve existing problems. However, it might be the case there isn’t a more efficient algorithm.

In summary, there are a number of techniques that can be used to try to solve time-consuming problems. Often additional computational power, additional computers, or a better algorithm (if one exists) can solve the problem. But not always. There are some problems, such as the traveling salesman problem, that by their very nature cannot be solved exactly for large values of  $n$  in a reasonable amount of time.<sup>7</sup>

---

<sup>6</sup>See <http://setiathome.berkeley.edu/>

<sup>7</sup>Still another approach is to obtain an approximate, rather than exact, solution. There are some problems that are impossible to solve exactly in a reasonable amount of time, but which have efficient algorithms that will provide an approximate answer (or an answer that is likely, but not guaranteed, to be correct or close to correct).

## 9.8 Problem Solutions

### 9.8.1 Introductory Problem Solutions

Here are model solutions to the questions at the start of this chapter.

1. Each item must be checked against each other item. So the first object would be checked against 99 others (note each object does not need to be checked against itself), the second against 98 others (since it's already been checked against the first), the third against 97 others, and so on until we have the sum

$$99 + 98 + 97 + 96 + \dots + 3 + 2 + 1 = \sum_{i=1}^{99} i.$$

Note this is a type of sum discussed in the last chapter on counting. In particular, using the formula  $\sum_{i=1}^{n-1} i = (n-1)n/2$  gives  $99(100)/2 = 4950$ .

2. This is similar to the previous question except the sum is now  $\sum_{i=1}^{199} i = 199(200)/2 = 19,900$ .
3. The ratio is  $19900/4950$ , or roughly 4.
4. (a) Analogous to the explanation in Problem 1, the answer is

$$(n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1 = \sum_{i=1}^{n-1} i = (n-1)n/2.$$

(b) If you double  $n$ , the number of times is  $(2n-1)(2n)/2$  (just substitute  $2n$  for  $n$  in the part (a) answer). So the ratio is

$$\frac{(2n-1)(2n)/2}{(n-1)n/2} = \frac{(2n-1)(2)}{n-1}.$$

Notice that this is roughly 4.

Here are two additional problems to think about:

- Suppose that in Problem 4b that  $n$  triples instead of doubles. What would the answer be then? How about if it increases by a factor of 10?
- Explain in Problems 3 and 4 why computer scientists would be more interested in the approximate answer (“roughly 4”) rather than the exact answer.

## 9.8.2 Example Problem Solutions

Here are solutions to the example problems from Section 9.5.

1. (a) In lines 1 – 4 the outer loop is executed  $n$  times, as is the middle loop and the inner loop. Since these loops are nested, the number of operations is  $n^3$ . In lines 5 – 7 both the outer loop and inner loop are executed  $n$  times, for a total of  $n^2$  operations. Since the loops in lines 5 – 7 are executed after the loops in lines 1 – 4 are completed, the total number of operations is  $n^3 + n^2$ . Since  $n^3$  grows more quickly than  $n^2$ , this is written as  $\Theta(n^3)$ .

(b) In lines 1 – 4 the outer loop is executed  $n$  times, and the middle loop and the inner loop are each executed 3 times. Since these loops are nested, the number of operations is  $9n$ . In lines 5 – 7 both the outer loop and inner loop are executed  $n$  times, for a total of  $n^2$  operations. Since the loops in lines 5 – 7 are executed after the loops in lines 1 – 4 are completed, the total number of operations is  $9n + n^2$ . Since  $n^2$  grows more quickly than  $9n$ , this is written as  $\Theta(n^2)$ .

(c) In lines 1 – 4 the outer and inner loops are executed  $n$  times, and the middle loop is executed 3 times. Since these loops are nested, the number of operations is  $3n^2$ . In lines 5 – 7 both the outer loop and inner loop are executed  $n$  times, for a total of  $n^2$  operations. In lines 8 – 10 the outer and inner loops are also both executed  $n$  times each, for a total of  $n^2$  operations. Since the loops in lines 1 – 4 are executed first, then the loops in lines 5 – 7 are executed, then the loops in lines 8 – 10 are executed, the total number of operations is  $3n^2 + n^2 + n^2 = 5n^2$ . This is written as  $\Theta(n^2)$ .

(d) In lines 1 – 6 the outer loop and middle loop are each executed  $n$  times. As discussed in Section 9.4, the inner loop is executed  $\lg n$  times. Since these loops are nested, the number of operations is  $n^2 \lg n$ . In lines 7 – 9 both the outer loop and inner loop are executed  $n$  times, for a total of  $n^2$  operations. Since the loops in lines 7 – 9 are executed after the loops in lines 1 – 6 are completed, the total number of operations is  $n^2 \lg n + n^2$ . Since  $n^2 \lg n$  grows more quickly than  $n^2$ , this is written as  $\Theta(n^2 \lg n)$ .

2. (a) Since the algorithm is  $\Theta(n^2)$ , when the problem size doubles the time needed will increase by a factor of roughly  $2^2 = 4$ . So the program would take about  $4 \times 7 = 28$  minutes to run.

(b) When the problem size increases tenfold, the time needed increases by a factor of roughly  $10^2 = 100$ . So the program would take about  $100 \times 7 = 700$  minutes to run.

(c) Note here the problem size for each database is not increasing. Instead you are running the program on 20 same size databases. So the total time is  $20 \times 7 = 140$

minutes. (Make sure you can explain why the answer is  $20 \times 7 = 140$  minutes rather than  $20^2 \times 7 = 2800$  minutes.)

(d) Running the program on 20 databases each of 10,000 records takes considerably less time.

(e) The original time is 9 minutes and the problem size has increased tenfold. Since the algorithm is now  $\Theta(n^3)$ , a tenfold increase in problem size means roughly a  $10^3 = 1000$  increase in time. So the total time is roughly 9000 minutes, which is six and a quarter days.

(f) The problem size increases by a factor of 5, so the time would increase by  $5^2 = 25$  due to this. However, the computer speed increase would decrease the time by a factor of 5. So the total time would be roughly  $(5^2 \times 7)/5 = 35$  minutes. (Make sure you can explain why the time is 35 minutes rather than 7 minutes.)

3. (a) In the case where a large difference is found between  $a[1,1,1]$  and  $a[2,1,1]$ , Line 9 is executed only once.

(b) It is possible for the algorithm to run all the way through the triple loops. This will happen if no large difference is found, or if a large difference is found on the final comparison. In either case, the outer loop is executed  $n - 1$  times (note that loop goes from 1 to  $n - 1$  because of the  $i + 1$  in Line 9), the middle loop  $n$  times, and the inner loop  $n$  times. So the total is  $(n - 1)(n)(n) = n^3 - n^2$ .

(c) The fewest number of times is still 1, so the ratio is  $1/1 = 1$ .

(d) Now the outer loop is executed  $2n - 1$  times, the middle loop  $2n$  times, and the inner loop also  $2n$  times. So the total number of times is  $(2n - 1)(2n)(2n) = 8n^3 - 4n^2$ . The ratio is therefore  $(8n^3 - 4n^2)/(n^3 - n^2)$ , or roughly 8.

## 9.9 Some Further Questions

Here are some further questions to think about:

1. What does the topic of algorithmic complexity tell us about computer science as a field? About how computer scientists think, and about how they solve problems?
2. The function  $f(n) = n!$  grows very quickly. Suppose someone says  $g(n) = n! + 1$  grows even more quickly. Are they correct? How about  $g(n) = 2 \times n!$ ?
3. Do you think there is a function that grows more quickly than all other functions? Why or why not?
4. Suppose someone says that as computers get faster and faster all computational problems will be solvable in a very short amount of time. Do you agree or disagree?

5. Give examples, other than those given in this chapter, where the issue of how processes scale is important. If possible, give examples from areas that you are interested in outside of computer science.
6. Give examples, other than those given in this chapter, where there are massive amounts of data being gathered or generated. If possible, give examples from areas that you are interested in outside of computer science.

# Chapter 10

## What is Computer Science?

Computer science  $\neq$  computer programming.

### 10.1 Introduction

#### 10.1.1 Introductory Problem

Suppose someone claims “software engineering” is the same as “computer science.” Do you agree or disagree, and why?

#### 10.1.2 Overview

What is computer science? If you ask this to a number of people, you will get a variety of answers. When some people think of computer science, they often think of computer programming. And while computer programming is an important part of computer science, it is not the same as computer science. For instance, there are many computer scientists who do not do program. They might be theoreticians, analysts, or administrators. In this chapter we’ll examine three related questions:

- What is computer science?
- What is the difference between computer science and computer programming?
- How is computer science similar to, and different from, related fields such as computer engineering?

The purpose of this chapter is therefore to clarify what computer science is and what its relation to similar fields is. The discussion in this chapter leads into the material in the next two chapters, where we’ll look at the software development side of computer science in more detail, and we’ll see the basics of a specific programming language, Python.

### 10.1.3 Motivation

To understand computer programming and computer science it is important to know what they are, to clarify some common misperceptions, and to understand their similarities to and differences from related areas.

### 10.1.4 Goals

Upon completing this material, you should be able to do the following:

1. Be able to explain, in your own words, what computer science is.
2. Be able to explain what fields such as computer engineering are, and how they are related to, but different from, computer science.
3. Be able to explain how computer science differs from computer programming.

### 10.1.5 Connection with Mathematics, and with Technology and Society

This chapter does not have a direct tie to mathematics or to technology and society. However, it will be useful background information for when we look at topics such as software development and computer programming, where there is a stronger connection to the liberal education requirements.

## 10.2 What is Computer Science?

Throughout this class we have been exploring different aspects of computer science such as the role of algorithms, and how computers represent and operate on different types of data. From this material you should have gotten a good amount of exposure to computer science. And so it might seem odd to raise the question “What is Computer Science?” at this point. However, now that we are looking at the areas of computer programming and software development in this class, it is a good time to explore more precisely what computer science is, especially with respect to related fields.

If you look up definitions of “computer science,” you will find a number of different possibilities:

- Merriam-Webster Online Dictionary: “a branch of science that deals with the theory of computation or the design of computers.”<sup>1</sup>

---

<sup>1</sup>From <http://www.merriam-webster.com/dictionary/computer%20science>; accessed Dec. 31, 2013.

- Cambridge Dictionaries Online: “the study of computers, how they work, and how to make use of them ...”.<sup>2</sup>
- Webopedia: “The study of computers, including both hardware and software design ...”.<sup>3</sup>
- Wikipedia: “Computer science (abbreviated CS or CompSci) is the scientific and practical approach to computation and its applications. It is the systematic study of the feasibility, structure, expression, and mechanization of the methodical processes (or algorithms) that underlie the acquisition, representation, processing, storage, communication of, and access to information ...”.<sup>4</sup>

As the differences among these examples indicate, there are difficulties in defining precisely what computer science is. The remainder of this section will discuss in more detail some of these difficulties.

One challenge is that computer science consists of a number of subfields or subparts. The definitions above mention the theory of computation, computer design, how computers work, how to use computers, hardware and software design, applications, and other subparts. Each of these is important. However, one mistake people often make is to equate computer science with just one subpart. For example, computer programming is an important part of computer science, but it is not all of computer science. Similarly, computer use, such as using databases, spreadsheets, webpage design tools, etc. is often something computer scientists do, but is only a small part of computer science.

There are various ways to subdivide computer science. One possible distinction is between hardware and software. Another is between theoretical computer science and applied computer science. Still other subpart classifications are more detailed and/or divide the subparts even further. For example, software design can be broken into system software such as operating systems and compilers, and application software such as graphics or database software. Still other classifications provide additional subareas; for example here is a subarea list that contains some areas already mentioned and some additional ones:<sup>5</sup>

- systems
- theory
- artificial intelligence/robotics
- software engineering/programming languages

---

<sup>2</sup>From <http://dictionary.cambridge.org/us/dictionary/business-english/computer-science?q=computer+science>; accessed Dec. 31, 2013.

<sup>3</sup>From [http://www.webopedia.com/TERM/C/computer\\_science.html](http://www.webopedia.com/TERM/C/computer_science.html); accessed Dec. 31, 2013.

<sup>4</sup>From [http://en.wikipedia.org/wiki/Computer\\_science](http://en.wikipedia.org/wiki/Computer_science); accessed Dec. 31, 2013.

<sup>5</sup>These are the upper division elective “tracks” that University of Minnesota-Twin Cities Computer Science B.S. students can choose from to fulfill their degree requirements.

- computational science
- graphics and visualization
- architecture and hardware systems
- bioinformatics and computational biology
- databases
- graphical information systems
- human computer interaction
- security
- networks
- software and data systems development
- big data

A second challenge is that many subfields of computer science are interdisciplinary. For example, robotics draws not only on computer science, but also on other fields such as electrical engineering and mechanical engineering. Artificial intelligence overlaps with the fields such as psychology and linguistics. Computer hardware research overlaps heavily with electrical engineering and material science. Computer graphics overlaps with a variety of areas including optics, image processing, perception, and medical imaging; these in turn are subareas of other fields such as physics, psychology, and medical research. In short, many subfields of computer science overlap with, draw upon, and contribute to other fields.

A third challenge is that computer science is closely related to other fields such as software engineering or computer engineering. Sometimes these other fields are seen as subfields of computer science, sometimes as distinct, but closely related fields. The similarities and differences are important enough that we will discuss this point further in the next section.

A final challenge is that computer science is a rapidly evolving area, and so what constitutes computer science continues to change. In the early days of computing topics such as numerical computation and language specification were prominent; topics such as networking or “big data” were non-existent or not as important.

### 10.3 Closely Related Fields

As the last section mentioned, computer science is closely related to fields such as software engineering and computer engineering. And sometime these fields are seen as subfields

of computer science, and sometime they are seen as closely related fields. For example, some universities have separate programs in computer science, computer engineering, and software engineering; others might have just a single degree (usually computer science) but might have computer engineering and/or software engineering as emphases within that computer science degree.

Here is a commonly used diagram, which is explained further below, showing the relation between computer science and five related fields.

EE ---- CompE ---- CS ---- SE ---- IT --- IS

‘CS’ stands for computer science. Here is a brief explanation of the five other fields:

- *EE*, or *electrical engineering* is the study and practice of working with electricity, electronics, electromagnetism, and electrical systems. It consists of a number of subfields, some of which, such as power systems, are less directly related to computer science, and others, such as circuit design, that are more closely related. Most large universities (and many smaller ones) have an electrical engineering department.
- *CompE*, or *computer engineering* is the study and practice of designing, creating, and manufacturing computer hardware and components, especially microprocessors. Computer engineering is therefore closely related to both electrical engineering and computer science. Not all universities have a computer engineering degree. Some that do have it as a standalone program; however, often it is housed in the electrical engineering or computer science department.<sup>6</sup>

Computer engineering students take many of the same introductory classes as computer science students and electrical engineering students — beginning programming, introductory computer science theory, circuits, microcontrollers, etc. — and then take advanced courses in areas such as computer architecture and systems programming.

- *SE*, or *software engineering* is the study and practice of developing software. Software engineering is different from computer science because of its narrowed but deeper focus. For example, a student getting a software engineering degree would take many of the same classes as a computer science student, including programming, theory, and applications classes, but would take numerous advanced classes in software development. (The software development process is described in more detail in the next chapter.)

Some universities have separate software engineering programs. For most universities however, students wishing to get a software engineering background would get

---

<sup>6</sup>Or both. For example at the University of Minnesota-Twin Cities, Computer Engineering is a joint program of both the Electrical and Computer Engineering Department and the Computer Science and Engineering Department.

a computer science degree with a software engineering emphasis.<sup>7</sup>

- *IT or information technology*: The *ACM 2008 Curricular Guidelines for Information Technology*<sup>8</sup> defines IT as the “the computer technology needs of business, government, healthcare, schools, and other kinds of organizations.” It further describes the field as follows:

“its emphasis is on the technology itself more than on the information it conveys. IT is a new and rapidly growing field that started as a grassroots response to the practical, everyday needs of business and other organizations. Today, organizations of every kind are dependent on information technology. They need to have appropriate systems in place. These systems must work properly, be secure, and be upgraded, maintained, and replaced as appropriate. Employees throughout an organization require support from IT staff who understand computer systems and their software and are committed to solving whatever computer-related problems they might have. Graduates of Information Technology programs address these needs.”

The University of Minnesota College of Continuing Education has a degree in Information Technology Infrastructure (ITI). Students in this program take many computer science classes, some business classes, and electives that focus on technology infrastructure; these include advanced network, computer systems, and database classes among others.<sup>9</sup>

- *IS or information systems*, is more concerned with the business side of information technology. For example, questions such as “how could a new computer system make a business’s practices more efficient?” and “what type of computer system does a business need to manage its data?”, while they have computer science aspects, belong more to the field of information systems.<sup>10</sup> A student getting a degree in

---

<sup>7</sup>At the University of Minnesota-Twin Cities there is not a separate bachelor’s degree in software engineering. However, there is a Masters of Science in Software Engineering (MSSE). This is a professional degree for people who work in the local software industry.

<sup>8</sup>Available online at <http://www.acm.org/education/curricula/IT2008%20Curriculum.pdf>; accessed Jan. 9, 2014.

<sup>9</sup>More information on the ITI degree is online at <http://cce.umn.edu/BAS%2DIT%2DInfrastructure/>.

<sup>10</sup>The *ACM 2010 Curricular Recommendations for Information Systems*, available online at <http://www.acm.org/education/curricula/IS%202010%20ACM%20final.pdf>, provides a more formal and detailed description of this field:

“Information Systems as a field of academic study encompasses the concepts, principles, and processes for two broad areas of activity within organizations: 1) acquisition, deployment, management, and strategy for information technology resources and services (the information systems function; IS strategy, management, and acquisition; IT infrastructure; enterprise architecture; data and information) and 2) packaged system acquisition or system development, operation, and evolution of infrastructure and systems for use in organizational pro-

information systems would take many business classes such as project management and information system acquisition. They would also take classes in areas such as beginning programming, databases, and other applications that are critical to business systems and operations. In addition, they would take classes in areas such as information security, information system architecture and infrastructure, and e-commerce that involve both computer science and business. Information systems programs are usually housed within the business school.<sup>11</sup>

The Association for Computing Machinery, one of the major computer professional groups, has curricular recommendation reports for five of the six fields mentioned above (all except electrical engineering). These detailed reports, whose intended audience includes educators, accreditors, and professionals from related industries, are available online at <http://www.acm.org/education/curricula-recommendations>.

The list above is not comprehensive. There are other fields such as mathematics that have contributed to, overlap with, and are influenced by computer science.<sup>12</sup> There are also other specialized fields that are closely related to computer science. For example some colleges or universities have degrees in computer forensics.

In general, computer science is very broad in terms of what types of jobs a student with a computer science degree might go into. Many computer science students do go into programming or software development positions. Others work in application areas such as databases or artificial intelligence. Others might work in areas such as systems administration or networking. Still others will work in other subareas of computer science.

Computer science and the related degrees are also very fluid. Many computer professionals come from backgrounds in electrical engineering, mathematics, or other related areas, or even unrelated ones. In part this is because of the dynamic nature of computer science, and in part because computer science both relies on and contributes to so many other fields.

---

cesses (project management, system acquisition, system development, system operation, and system maintenance). The systems that deliver information and communications services in an organization combine both technical components and human operators and users. They capture, store, process, and communicate data, information, and knowledge.”

<sup>11</sup>At the University of Minnesota-Twin Cities, the the Carlson School of Management has a bachelors degree in Management of Information Systems (MIS). The University catalog describes the program:

“The management information systems (MIS) major prepares students to be leaders in conceptualizing, prescribing, developing, and delivering leading-edge information system applications that support business processes and management decision making. It provides students with an understanding of the functions of information systems in organizations and detailed knowledge of information system analysis, design, and operation.

<sup>12</sup>In fact many smaller schools house mathematics and computer science programs in the same department.



# Chapter 11

## Software Development

Programming “in the small”  $\neq$  programming “in the large.”

### 11.1 Introduction

#### 11.1.1 Introductory Problem

Mark each of the following as “true” or “false” and briefly justify your answer.

1. The largest software systems consist of hundreds of thousands of lines of code.
2. Although computer errors have led to significant negative effects, such as billions of dollars of business losses and mission failure for interplanetary space probes, they have never been a major contributing factor in a human fatality.
3. Although it has not always been the case, most large software projects are completed on time and under or at budget.

#### 11.1.2 Overview

Computer systems play an important part in our everyday life. But what do we know about those systems? How big are they? How reliable are they? How are they constructed? Are most attempted software projects successful? The aim of this chapter is to examine these and other aspects of computer software and computer programming. In particular, this chapter provides a “big picture” discussion of software development and computer programming.

#### 11.1.3 Motivation

In 1990 the Mars Climate Orbiter entered the Martian atmosphere at the wrong angle, resulting in a catastrophic failure. The cause was traced to a software miscommunication:

one group working on the software was using metric units of thrust, another was using English units.

Large software systems are important in many aspects of society. Here are a few examples:

- As the introductory example shows, software systems are an important (and sometimes faulty) part of space exploration.
- Telecommunications software is an important part of the mobile phone infrastructure.
- Office productivity software such as word processors, spreadsheets, and database software are large programs used by many individuals and businesses.
- Businesses also rely on a number of software systems such as payroll and inventory systems.
- As was briefly discussed in a previous chapter, system software such as operating systems and compilers are programs that allow computers to do system-related tasks such as translating high level programming code to machine instructions.
- Computer games are rarely simple programs, but are often complicated systems involving graphics; user interaction; a rich database of plots, characters, and game objects; ability to play over a network with multiple players; etc.

In summary, software systems play an important role in many, many important areas of society whether the areas are scientific, business-related, industrial, governmental, communications-related, recreational, or personal. These systems allow individuals, businesses, government agencies, and other organizations to perform tasks that would otherwise be impossible or difficult. But the systems can also create problems if they malfunction. To better understand the role of software it is important to know some fundamentals about software and its development.

#### 11.1.4 Goals

Upon completing this chapter, you should be able to do the following:

1. Be able to explain some of the key differences between writing a small computer program and a large one.
2. Be able to state some fundamental facts about software, software projects, and the software development process.
3. Be able to explain the main steps in the software development process.

### 11.1.5 Connection with Mathematics, and with Technology and Society

Software development has a number of important connections with mathematics. On one hand, programming has a strong connection with mathematics as much of it involves computations in some way, shape, or form. On the other hand there are deep and important connections between software development and mathematics. However, many of these connections are advanced.<sup>1</sup> So — other than looking below at a problem involving counting and software testing — we will not explore the connection between software development and mathematics further.

The connection between software systems and technology and society is more evident. As discussed in the motivation subsection above, software systems play a role in many important areas of society.

## 11.2 Programming “In the Large” Versus “In the Small”

One common view of computer programming is similar to a common view of how writers work: they work solo at their keyboard for many hours and finally create a significant work. The writer might create a book, and the programmer an important computer program.

While this view is sometimes correct, it does not reflect the reality of most programmers. Why not? There are a number of reasons, including the following:

- Most programmers work on multiperson projects rather than working alone.
- While programming sometimes consists of writing new code, it also involves correcting, updating, etc. old code, probably code that other programmers have written.
- There is more to creating computer software than coding. For example, it is often not clear exactly what the program needs to do, so part of the software development process is clarifying what the requirements for the software are.
- A program is often not “standalone;” it might itself be part of an even large computer program, which in turn might need to interact with still other computer programs.

The view of a single programmer working on a well-defined project and producing an entire program is an example of what is called “programming in the small.” However, large important systems are the work of many people working in many places over a long time span. Much of the software we rely on, whether it be office productivity software such as spreadsheets and word processors; web browser software; operating systems;

---

<sup>1</sup>For example, one use of *formal methods* is to try to prove mathematically that a code section is correct.

business software systems that track and analyze inventory, orders, and cost; software for telecommunication systems; etc. are large pieces of software. For example operating systems such as Linux/Unix, Windows, and MacOS consist of millions of lines of code.

Working on a large multiperson system is significantly different from writing a program that might be used only by a handful of people, perhaps only the programmer himself or herself. Large systems will have many users, might go through many versions, and might involve special safety, security, accuracy, or other features.

### 11.3 Parts of the Software Development Process

Software development requires more than only programming or coding. As the last section mentioned, software development often involves additional, related steps:

- *Requirements Engineering*: It might seem odd that in most large systems the first challenge is defining, and defining precisely, what the system is to do. Why does this challenge even exist? Isn't there usually a well-defined problem or set of tasks, and what we are looking for is a computer program to solve that problem or do those tasks?

However, for large systems it is rarely if ever the case that the problem or task is completely and precisely defined. Let's look at a very simple example: suppose someone asks you to write a computer program to sort a list. What does this mean? Sort in ascending or descending order, or allow the user to choose either? Sort a list of numbers? Names? Addresses? Any type of list involving alphabetical or numeric data? If the list involves both alphabetical and numeric data should numbers appear before characters or vice versa? How about sorting upper case vs. lower case? And are there any restrictions or special requirements? For example, is there any restriction on the list size or the number of characters in each list item? Should there be a check for obviously bad data? If so, how should it be handled? What type of a device should the program run on? Should the program be able to be called by other programs? If so, what form should the list be in to ensure both programs can work with it? Would the sorted list overwrite the original list (so the original list would no longer be around), or would that be a problem?

So even for this very simple task there are numerous clarifying questions we need to ask before writing the program. Defining the "requirements," or precisely what a system is to do, is difficult for a number of other reasons as well. These include the following:

- Large systems are complicated and it is difficult to specify all the requirements accurately and clearly.
- Clients often don't know exactly what they want. They might have a high-level idea, but are unsure about the details. They might not know what is possible.

Or they might have multiple divisions within their organization, and not know what all the different users want out of the system.

- There are often tradeoffs involved in software systems. For example, a more complicated system might have more features and capabilities, but take longer to develop. A more secure system might be harder for people to use. A nonessential feature might be desirable but also costly. A new user interface might be more efficient than the interface of an old system, but it will take additional time for users to learn.
  - Changing landscapes. The requirements might (in fact, for a large project, will) change over time. An organization’s needs, processes, or infrastructure might change. For example, suppose a company wanted some software that needed to communicate with the company’s existing databases, and that the company had multiple locations, each with its own database. However, due to company restructuring as well as falling telecommunications costs and increasing telecommunications abilities the company decides to consolidate all its databases into a single location. Then the software would only need to communicate with this single database rather than with multiple ones.
- *Design*: Suppose you are building a house. You don’t begin by actually starting the construction. Instead there is a planning and design phase first. Similarly, large software systems need careful design: What is the system’s “architecture,” for example what are its main parts and how are they related? What are the subparts of the main parts? How does the system interact with any related systems it must communicate with? Etc.
  - *Coding*: This is the part of the development process that most people think of when they think of software development: it is the part consisting of actually writing the code. If the requirements are well-specified and stable, and if there is a good, informative design, then coding should actually be the easiest part of the development process.
  - *Testing*: No people, not even the best programmers, write substantial amounts of code error-free the first time. A part of a programmer’s job is to test the code he or she writes. However, testing is complicated enough and important enough that it constitutes a separate part of the development process. In fact, many development companies have different people test the software than write it (think about why this is useful). Some large development companies have an entire group whose job is testing.

Why is testing nontrivial? Here are a few of many reasons:

- It is usually impossible to check all possible cases. For example, suppose a program asks a user to input a text string of 20 characters or fewer. There

are so many possible names that it is impossible to check all of them in any reasonable amount of time. (See the next section for a related problem.)

- Programs must guard against erroneous and unexpected events. Suppose a program asks a user to input a temperature. What happens if they mistype 98.6 as 98,6 or as @iu.t@? Tasks such as checking input add complexity to programs, and are often difficult to do rigorously.
- Many program errors are not within a piece of code itself, but in how the piece of code interacts with other pieces of code in a larger system.
- *Maintenance*: The word “maintenance” lacks glamor. Would you really want maintenance to be a significant part of a successful software project? Actually you would. Part of maintenance is adding additional features to a successful piece of software. Part of it is fixing obscure defects or security issues that wouldn’t have been found without a large user base. Part of it is updating successful software from an older version to a newer version. Part of it is modifying the software so it is useful on a variety of devices (e.g., perhaps on tablet computers or smartphones).

In short, think of software maintenance like you think about automobile maintenance or house maintenance. You do not want to spend too much time or money on maintenance. However, if you have a car or house you really like you do preventive maintenance to keep it in good shape, and fix occasional problems that occur. And occasionally you do major “maintenance” such as adding a room to a house. Similarly, a reasonable and long-term commitment to maintenance is a sign of a successful software product, not an unsuccessful one.

A more comprehensive discussion of software development would discuss the above items in more detail, as well as identifying other parts of software development. However, the items explained above illustrate the key point that software development is more than just coding, and that developing large software systems is different than developing smaller systems or writing a single short program.

The steps above (sometimes with additional steps included) are sometimes put in a sequence and presented as the “waterfall” model of software development: first you specify the requirements, then you design the system, then you do the coding, then you test the system, and afterwards you do system maintenance. This is a useful model in that it identifies these key parts of the development process. However, it should not be taken as a representation of how development actually occurs in practice. For example, as mentioned above, requirements sometimes change in the middle of the development process, and so one cannot specify the requirements and assume that stage is entirely finished. As another example, often during the design or coding it becomes clear that certain requirements were omitted, or incorrectly or ambiguously specified; or during the coding phase a better design of some subparts of the system might be discovered. Additionally, development often proceeds iteratively. For example, certain core requirements might be specified, and the related part of the system designed, coded, and tested; then a second iteration would

specify requirements for an additional portion of the system, and it in turn would be designed, etc.; and in general the system would be built through a number of iterations.

## 11.4 Software Challenges and Risks

The inherent complexity of large software systems presents a number of challenges. Large systems consist of many, many parts and many, many subparts. The largest systems consist of millions of lines of code. It is impossible for any single person to understand the entire system in depth, or even a substantial part of it.

Because of this, large systems are difficult to design, implement, and maintain. In fact, large systems do not have a large success rate. Although software development has improved and is continuing to improve, large systems are more often than not delivered late and/or over budget.<sup>2</sup>

Moreover, even delivered software is not error-free. Rates of 1 or 2 errors per thousands of delivered lines of code are common. And while this might not seem like a large error rate, in a system of hundreds of thousands if not millions of lines of code it results in a large number of errors. Many of these errors are minor, but occasionally a “small” error can have serious or catastrophic results. Software errors have contributed to mission failures such as in the Mars Climate Orbiter example, financial losses that yearly total in the billions of dollars, or even human injury or loss of life, as in the Therac-25 incident we’ll (probably) explore in class.

Because certain systems such as space exploration software, medical software, military software, or financial software have special requirements, they need extra care, know-how, and oversight in their development. Careful software processes can significantly decrease the number of errors in software systems; however, this requires additional costs and know-how.

## 11.5 Some Example Problems

**Problem 1:** Suppose a software system is 200,000 lines long and is estimated to contain 2.8 errors per thousand lines of code. How many errors does this mean the code is likely to contain?

**Problem 2:** Suppose a program requires a user to enter an 8-character string, where each character in the string can be a lower-case alphabetic character, and upper-case alphabetic character, or a digit 0–9. Characters can be repeated. Suppose further that you wish to test the code by checking all possible bitstrings, and that your test code can check 10,000 bitstrings per second. How long will it take for your code to test all of the possible 8-character strings?

---

<sup>2</sup>The Oct. 2013 ACA site failure is just one of many examples.

**Problem 3:** In testing it is desirable (but not always easy or even possible) to test every “path” through the code. Consider the following algorithm fragment. In the fragment the “...” represent lines that are omitted but which might provide output or modify the variable values, but do not include any `if` or `else` statements. How many different paths are there through the algorithm? For example if `violinCount` is 5, `violaCount` is 20, and `celloCount` is 10, then the algorithm will execute different statements than when `violinCount` is 8, `violaCount` is 12, and `celloCount` is 4. So these two cases represent two different paths.

```
if violinCount > 10
    if celloCount < 10
        ...
    else if celloCount < 20
        ...
    else
        ...
else
    ...
if violaCount equals celloCount
    ...
else
    ...
if bassCount < 10
    if doubleBassoonCount < 10
        ...
    else
        ...
```

**Problem 4:** Explain why software development is more than only doing computer programming.

## 11.6 Problem Solutions

### Introductory Problem

- (1) False; they are larger, consisting of millions or tens of millions of lines of code.
- (2) False; for example, both software and hardware malfunctions were contributing factors in the Therac-25 incidents (which will likely be discussed in class).
- (3) False; although the situation is improving, recent studies have shown only about one third of large software projects are completed on time and don't exceed their budget.

**Problem 1**

If a program is 200,000 lines long and contains an average of 2.8 errors/thousand lines of code, then it contains  $2.8 \times 200 = 560$  errors.

**Problem 2**

The 8-character string has 62 possibilities for each character. Characters can be repeated and order matters, so there are  $62^8 = 218,340,105,584,896$ , or about 218 trillion possible strings. If 10,000 of these can be processed per second, then it will take the program  $218,340,105,584,896/10,000$ , or approximately 21,834,010,558, seconds. This is about 22 billion seconds, which is almost 7 centuries.

**Problem 3**

Notice that the program will first take one of the four paths through the first part of the code:

```
if violinCount > 10
    if celloCount < 10
        ...
    else if celloCount < 20
        ...
    else
        ...
else
    ...
```

Then it will take one of two paths through the next part:

```
if violaCount equals celloCount
    ...
else
    ...
```

Finally, it will take one of three paths through the final part (note that in addition to the two paths indicated by the dots, there is a third path when `bassCount < 10` is False):

```
if bassCount < 10
    if doubleBassoonCount < 10
        ...
    else
        ...
```

So there are a total of  $4 \times 2 \times 3 = 24$  different paths.

**Problem 4**

Answers can vary. One possibility is mentioning that, in addition to computer programming, software development involves requirements engineering, software design, testing, and maintenance.



# Chapter 12

## Python Reference

“All programmers are playwrights and all computers are lousy actors.”<sup>1</sup>

### 12.1 Introduction

This chapter is a quick introduction to Python. It will have a different structure than other chapters, presenting some Python along with examples. This chapter is not comprehensive: Python, like most computer languages, is extensive, and the Python programming language has been the topic of many entire books. So the material in this chapter is only an introduction that will focus on the basics of Python you need for the class.

This material is meant to complement online references and the in-class presentation of Python. Why have a chapter on Python if there are other resources? It is often useful to have additional or alternative explanation of programming language concepts, as well as additional examples. The Python site <http://www.python.org> contains additional reference material, as well as an online tutorial. Plan on using that site extensively. The class lectures will present additional examples, some of which will involve more complicated use of Python than in this introductory chapter. So in your study of Python you should also attend those classes and study the additional examples.

You should be able to run any of the examples in this section on the Python interpreter.<sup>2</sup> Recall that on the lab machines used for this class you can run the interpreter simply by logging in and typing `python` in a terminal window. More detailed instructions about using the Python interpreter are below.

As a final note, the best way to learn about computer programming is to program. So rather than just reading this chapter on its own, you are encouraged to try the examples in this chapter in the Python interpreter, vary them and see what happens, make up your own examples, etc.<sup>3</sup>

---

<sup>1</sup>From <http://www.cs.cmu.edu/~pattis/quotations.html>; accessed Dec. 18, 2013.

<sup>2</sup>The Python in this chapter uses version 2.7.

<sup>3</sup>This chapter contains only a few problems; additional problems are in the next chapter.

## 12.2 Getting Started with The Python Interpreter

To use Python on the CSE lab machines, you can type `python` after opening a terminal window. This should put you in the interpreter, and you should see a `>>>` prompt. (If this doesn't work, see a TA.) At the Python prompt you can type in Python statements. Here is an example, including what happens if you make a syntax error:

```
>>> snowDepth = 5
>>> print snowDepth
5
>>> set beehiveCount = 2
      File "<stdin>", line 1
        set beehiveCount = 2
          ^
SyntaxError: invalid syntax
>>> beehiveCount = 2
>>> print beehiveCount
2
```

To exit the interpreter, type `CNTL-d`, that is, press the `control` and (lower case) `d` at the same time.

Section 12.16 below will explain some additional use of the interpreter.

## 12.3 Basics: Data Types and Operations

An earlier chapter discussed data and data representation. How does Python deal with data? As you might expect from the earlier discussion, Python has many data types. In this section we'll cover a few of them that you'll use in this class.

### 12.3.1 Numbers

Recall from the chapter on data representation that computers distinguish between integers and floating point numbers. This is reflected in many languages, including Python: when working with numbers we often need to specify if they are integers or floating point numbers.

Decimal integers are represented by strings of digits with negative numbers preceded by a minus sign: `42`, `0`, and `-65536` are all integers. However, a few cautions: First, don't include commas in Python numbers. For example, the last number above was *not* written as `-65,536`. Here is the result from the Python interpreter if you do mistakenly use a comma:

```
>>> 2 * -65,536
(-130, 536)
```

Whatever is happening here, it is very likely not what we want. The second caution is don't include leading zeros in decimal numbers. For example, even though 042 base 10 is, strictly speaking, equivalent to 42 base 10, we likely wouldn't write the former. Python doesn't like the 042 either.<sup>4</sup> Third, don't include a trailing decimal point, such as writing 42 as 42. When Python sees a number with a decimal point, it interprets it as a floating point number:

```
>>> 2 * 4.  
8.0
```

If you wish, you can represent integers in binary or hexadecimal. To represent a number in binary, prefix it with an 0b. For example, 0b10 is binary for the decimal value 2. Prefix hexadecimal numbers with an 0x. As an example, 0x1af is the hexadecimal for 431 (since  $431 = 1 \times 256 + 10 \times 16 + 15 \times 1$ ). Note that the leading zero in these representations is telling the Python interpreter to be prepared for a non-decimal representation.

Floating point numbers differ from integers in two ways. First, floating point numbers have a decimal point followed by zero or more digits, for example 0.0 or 3.14159. Second, floating point numbers can be larger or smaller than integers. To accommodate this, Python allows floating point numbers to be written in scientific notation. This is done by putting an e between the mantissa and the exponent. For example, the Python floating-point number 5.7e4 is the Python representation for the number  $5.7 \times 10^4$ , which in turn is the same as the Python floating-point number 57000.0. Much larger numbers, such as 6.02e23, (in conventional scientific notation  $6.02 \times 10^{23}$ ), very negative numbers, such as -2.0542e45, ( $-2.0542 \times 10^{45}$ ), and very small numbers, such as 6.626e-34 ( $6.626 \times 10^{-34}$ ) are also possible.

### 12.3.2 Text

Working with text in Python is simpler than working with numbers. To indicate a string of text, simply enclose it in either single or double quotes. For example, 'King Canute' and "King Canute" are both the same string. Notice you can include spaces within the string.

One complication with text is that we sometimes want to include special characters in a text string. For example, what if we want to have quotation marks within a string? A string attempt such as "He said, "You did well."" does not work: it causes a syntax error in Python.

With quotation marks, you can include a single quote within a double-quote delimited string, or vice versa. For example, 'He said, "You did well."' does work. But what if you want both single and double quotes in the string? Or what if we want other special characters such as a tab or line break? A way to indicate special characters in Python is by prefixing them with a specially designated character, the backslash character \, as shown:

---

<sup>4</sup>Actually it doesn't object to it, but it interprets it as an *octal* or base 8 number.

```
>>> print "A line with a \t tab and a \n line break"
A line with a   tab and a
  line break
>>> print "A line with a single quote \', double quote \", and backslash \\"
A line with a single quote ', double quote ", and backslash \.
```

### 12.3.3 Booleans

Boolean variables are variables whose value can be `True` or `False`. They are often used to indicate the status of a condition, for example `leftValveOpen = True`. Remember to capitalize the first letter in `True` and `False`.

## 12.4 Variables

We have met variables before: they are items in algorithms or programs whose values can change. Variables in Python can be of many types, including integer, floating point, boolean, and text.

### 12.4.1 Variable Names

There are rules for what constitutes a valid variable name in Python. Variable names must start with an uppercase alphabetic character, a lowercase alphabetic character, or an underscore. This first character can then be followed by zero or more uppercase alphabetic characters, lowercase characters, underscores, or digits. So, for example, `dog`, `Cat`, `zebra87`, `_mandrillCount`, and `platypus_count` are all valid variable names, while `5dog`, `Cat!`, and `octop@us` are not.

As mentioned in an earlier chapter, it is better to have descriptive variable names, for example `baboonCount` rather than just `b`. This makes programs easier to read and understand. Python allows multiword variable names; it is good practice to separate the words by underscores (e.g., `baboon_count`) or use “camelCase,” that is, capitalizing the first letter of each word (other than perhaps the first). However, including blank spaces within variable names is not allowed.

### 12.4.2 Assignment and Use

A Python statement such as `threadCount = 500` assigns 500 to the variable `threadCount`. The left side of the assignment statement is a variable name, which is followed by the assignment operator `=`. The right hand side can involve values, operators, and/or variables. For example, each of the following are valid Python assignment statements:

```
molluskCount = 12 + 8 + 4
animalCount = zebraCount
```

```

animalCount = zebraCount + gnuCount + 2
lobsterCount = lobsterCount + 1

```

## 12.5 Operators

Python, like every computer language, includes a variety of operators. This section provides an overview of commonly used arithmetic, relational, and logical operators.

### 12.5.1 Arithmetic Operations

Suppose you are constructing a triangular garden plot. The triangular area measures 6 feet in one direction and 8 feet in the other. What is the square footage of the area? You remember the area of a triangle is one-half the length of the base times the height. You plug this into the Python interpreter and get the following:

```

>>> b = 6
>>> h = 8
>>> (1/2)*b*h
0

```

What went wrong?

Python contains the usual arithmetic operations of addition, subtraction, division, and multiplication. It also contains a variant of division (*floor division*) that always rounds down, a remainder operation, and an exponent operation, as shown in the following table:

Operation	Symbol	Example	Value
Add	+	3 + 5	8
Subtract	-	72 - 21	51
Multiply	*	7.2 * 4.5	32.4
Divide	/	7 / 3.5	2.0
Rounded Down Divide	//	7 / 3.2	2
Remainder	%	7 % 3	1
Exponent	**	7 ** 3	343

Most use of these operations is straightforward. However, as the example above illustrates, some further explanation and cautions are needed:

- There is an order of operations for doing arithmetic. For example, is the addition done before the multiplication in  $4 + 3 * 2$ , or is the multiplication done first? The rule is — absent any parentheses — multiplication before addition. However, you can avoid memorizing the order of operation rules by using parentheses. *Use parentheses.* It will save you from making some hard-to-find errors, and will make the code easier for you and others to read and understand.

- Division can be confusing. Specifically, what should  $7/3$  be? Should it be  $2.333333$ ?  $2\ 1/3$ ? Should we round down to 2? Round up to 3? Note part of the problem here is that 7 and 3 are integers; should the result be an integer or a floating point number? And what is the `//` division for? Here are a few examples along with explanations:

```
>>> 7/3
2
>>> -7/3
-3
>>> 7.0/3
2.3333333333333335
>>> -7.0/3.0
-2.3333333333333335
>>> 7//3
2
>>> -7//3
-3
>>> 7.0//3
2.0
>>> -7.0//3
-3.0
```

In the first test here we are doing integer division, and the result is an integer. The result is rounded down to the next lowest integer. The rounding is always down, regardless of whether any fractional part is greater than  $1/2$  or not. For example,  $8/3$  is 2, not 3. Also “rounded down” means what you expect when the result is positive: just drop any fractional part. However, as the second test shows, when the result is negative, “rounded down” means going more negative; so in the case of  $-7/3$  the result is  $-3$ , not  $-2$ .

The third and fourth tests show what happens when one or both numbers is a floating point number: the result is a floating point number. The rule with usual division, i.e., with the `/` operator, is that the result is an integer only if both numbers are integers, otherwise the result is a floating point number. This is true even if the result of dividing two floating point numbers comes out evenly. For example,  $6.0/3.0$  yields  $2.0$ ; note this is represented as a floating point number rather than as the integer 2.

The last four tests show the difference between division using `//` and division using `/`. When both numbers are integers, there is no difference: compare the fifth and sixth tests with the first two tests. However, when one or both of the two numbers is a floating point number, then `//` rounds down. Note the result is still represented as a floating point number, with a 0 to the right of the decimal point.

## 12.5.2 Relational Operators

Relational operators compare two items. Most of the operators in the table below will be familiar to you: greater than, greater than or equal to, etc. Note the result is a value `True` or `False`.

Operation	Symbol	Example	Value
Less than	<	4 < 4	False
Greater than	>	8 > 4	True
Less-or-equal	<=	4 <= 8	True
Greater-or-equal	>=	4 >= 8	False
Equality	==	4 == 5	False
Inequality	!=	4 != 5	True

Three notes:

- As with arithmetic operators, use parentheses to indicate the order of operations.
- The symbol for equality is *two* equal signs. This is to distinguish it from the assignment operator. A common mistake is to have a line such as the following, which results in the shown error message:

```
>>> if b = 6:
      if b = 6:
          ^
```

```
SyntaxError: invalid syntax
```

This error vanishes if you use `==`.

- You can also apply these operators to characters and strings. The ordering used is alphabetic order, but capitals are always less than lower case characters. Here are a few examples:

```
>>> print "a" < "c"
True
>>> print "Z" < "w"
True
>>> print "aardvark" == "crustacean"
False
```

## 12.5.3 Logical Operators

Python includes the logical operators (which we saw in a previous chapter) `and`, `or`, and `not`. These are usually combined with relational operations, for example

```
(4 < 5) or (8 <= 2)
```

is `True`. Here is a more elaborate example:

```
(areaA * 5 >= areaB ** 3) and (not(areaA <=areaB))
```

Note this expression can be rewritten as an equivalent expression without a `not`:

```
(areaA * 5 >= areaB ** 3) and (areaA > areaB)
```

## 12.6 Lists and Indexing

In our everyday life, we often deal with groups of items. So, not surprisingly, computer languages include ways to work with such collections. One of these is Python's list structure.

A list in Python is a sequence of values or elements that share a common name and can be processed as a unit, but where each element can also be accessed and processed individually. To indicate a list in Python, use square brackets, [ and ], around comma separated elements; for example, here is a list of numbers: [1066, 1492, 1776], and a list of colors: ["red", "green", "blue", "white", "black", "cyan", "magenta", "yellow"]. The items in the list can be any Python type, and you are allowed to mix types. For example, [3.2, "red", False], which contains different types of data (a floating point number, a string, and a boolean value) is a valid Python list. Moreover, you can have lists of lists: [[11, 12, 13], [21, 22, 23]] can be thought of as a table with two rows and three columns.

To access an individual element in a list, use the list name along with the element position. Let's suppose we set up the following list:

```
>>> pieFlavors = ["apple", "pumpkin", "blueberry", "cherry", "pear"]
```

And suppose further you want to access the first and third elements in the list. You can do so as follows:

```
>>> print pieFlavors[0], pieFlavors[2]
apple blueberry
```

Note the syntax: the list name followed by a position number enclosed in square brackets. However, there is an obvious discrepancy here: we want to print out the first and third elements in the list, but we are supplying position numbers 0 and 2. Why?

Python, like many computer languages, starts list indexing at 0, not 1. There are historical and practical reasons for this, but it is admittedly very confusing. So the first element in the list has index or position 0, the second has index 1, and in general the  $i$ th element has index  $i - 1$ . The last (i.e., rightmost) element has index equal to the list length minus one. For example, the `pieFlavors` list has 5 elements, so the index of the last one, "pear", is 4.

What happens if you use an index that is not in the list? For instance, what will `pieFlavors[10]` return, since there are not enough elements in the list? In this case, Python returns an error message:

```
>>> print pieFlavors[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

There are a number of other useful things we can do with lists, some of which are explained below, and some of which will appear in class examples.

As a final note in this section, indexing also works with strings and allows you to access individual characters within the string. Here is an example:

```
>>> currentColor = "slateGrey34"
>>> print currentColor[0], currentColor[4], currentColor[9]
s e 3
```

Again, remember the indices start at 0.

## 12.7 Input/Output

The pseudocode in other chapters had a `get` command, a way for algorithms to get input from the user (or from a file) as the algorithm was running. Similarly, the pseudocode had a way to output information. It is reasonable to expect Python to have similar capabilities. And indeed it does.

We've already seen the simplest way Python outputs information: by using `print`. So `print "This is a test"` prints that text string. Or `print theaterLightCount` prints the value of the variable `theaterLightCount`.

Input, however, is a little more complicated. If you think about input, you want to get the input and store it somewhere. So you will need to use an assignment statement. Here is an example of a user input line:

```
applicantName = raw_input("Please input your surname: ")
```

The Python term `raw_input` tells Python that it should get input from the user. The text string within the `raw_input` parentheses is a prompt the computer will print to indicate to the user that input is needed. Once the user inputs their surname, the computer stores it in the variable `applicantName`.

But there's one further complication: what type will the input be? In the example above the input is obviously a string. But what if a program is working with apartment numbers? You can probably think of some cases where the program would want to treat an apartment number as a text string, and some where it'd want to treat it as an integer (and perhaps even some where it would want to treat it as a floating point number). More

generally suppose, for example, the user inputs 42. Is this supposed to be a text string? An integer? A floating point number?

By default, Python considers the input as a text string. But what if you wanted 42 to be treated as an integer? How do we tell this to Python? Like most languages Python has conversion functions. To change a string to an integer you use the conversion function `int`. To change a string to a floating point number you use the conversion function `float`.<sup>5</sup>

Here are some examples that include some input:

```
>>> llamaCount = int(raw_input("Enter the number of llamas: "))
Enter the number of llamas: 5
>>> llamaCount = llamaCount * 2
>>> print llamaCount
10
>>> payRate = float(raw_input("Enter the current pay rate: "))
Enter the current pay rate: 12.25
>>> payRate = payRate + 0.27
>>> print payRate
12.52
```

One common mistake is to forget that input lines like those above need *two* closing parentheses at their end: one to close `raw_input` and the second to close `int` or `float`.

## 12.8 Control Structures

The chapter on pseudocode mentioned “sequence, selection, and repetition.” Recall that sequence means executing statements in order, one after another. Selection means executing a statement or block of statements only if a given condition is true. And repetition means executing a statement or block of statements multiple times. Python has a number of *control structures* implementing sequence, selection, and repetition.

The most basic control structure, sequence, is to do one thing after another. This is represented in Python by the order of statements in the program text. Selection and repetition have special keywords, however.

Not surprisingly, selection in Python uses `if` and its variants. Here are examples:

```
if elephantFoodSupply < elephantFoodNeeds:
    print "Order more elephant food"
if applePieNumber > cherryPieNumber and applePieNumber > pecanPieNumber:
    print "There are more apple pies"
    applePieNumber = applePieNumber - 1
```

---

<sup>5</sup>There is also a conversion function `str` that converts a number to a string. For example, suppose `apartmentNumber` holds an integer value. Then the Python statement `apartmentString = str(apartmentNumber)` takes that value, turns it into a text string, and places the result in the variable `apartmentString`.

The `if` statement takes a condition that evaluates to `True` or `False`. (Note, as in the second `if` statement, this can be a compound expression.) The condition is followed by a colon. This is followed by the *body* of the `if`: an indented statement or set of statements that will be executed if the condition is `True`.

**Problem 1:** List some common errors a programmer might make in writing the code above.

As with pseudocode, there are `if` variants. These include `if-else`, `if-elif`, and `if-elif-else`. Here are examples, starting with `if-else`:

```
if applePieNumber > quincePieNumber:
    print "There are more apple pies."
    applePieNumber = applePieNumber - 1
else:
    print "There are at least as many quince pies."
    quincePieNumber = quincePieNumber - 1
```

Depending on the relative values of `applePieNumber` and `quincePieNumber` this code will execute one or the other block of indented statements. Note the form of the `else`: its indentation is the same as the `if`, there is a colon after the `else` keyword, and the lines to be executed for the `else` are indented.

Here is another `if` variant, `if-elif`:

```
if applePieNumber > quincePieNumber:
    applePieNumber = applePieNumber - 1
elif quincePieNumber > 10:
    quincePieNumber = quincePieNumber - 1
```

Note `elif` is short for *else if*. The `elif` is indented the same amount as the `if`, has a condition followed by a colon, and has a line or block of lines that will be executed when the `if` condition is `False` and the `elif` statement is `True`. With an `if-elif` it is possible for the `if` block of indented statements to be executed, the `elif` block of indented statements to be executed, or neither block of indented statements to be executed (if neither the `if` nor `elif` conditions are `True`).

It is possible to have multiple `elif`s, and/or to combine an `if-elif` with a `else`:

```
if cardValue == 1:
    print "Ace"
elif cardValue == 2:
    print "Deuce"
elif cardValue == 13:
    print "King"
elif cardValue == 12:
    print "Queen"
elif cardValue == 11:
    print "Jack"
elif cardValue == 10:
    print "Ten"
else:
    print "Other card"
```

Next we consider repetition structures. Python includes both `while` and `for` loops. We begin with `while` loops, which are very similar to our pseudocode form. Here is an example:

```
i = 1
count = 0
while i <= 100:
    if i % 2 == 0 or i % 3 == 0:
        count = count + 1
    i = i + 1
print count
```

This code fragment will repeatedly execute the body of the `while` statement as long as the condition in the `while` line is true. Specifically, it executes it for `i = 1, 2, ..., 100` and counts the number of such `i` that are divisible by 2 or 3 (or by both). Note the form of the `while` loop: the `while` keyword, followed by a condition, followed by a colon (don't forget the colon). Then the body of the loop consists of one or more indented statements. Note in the example the `while` loop body includes an `if` statement, and so `count = count + 1` is therefore indented even further. The first statement after the end of the `while` loop body is indented at the same level as the `while` line.

The `for` statement provides a way to step through a number of items. One elegant example is stepping through each item in a list, as in the following example:

```
for a in [1, 3, 6, 10]:
    print a, a*a
```

The indented body of the statement is executed repeatedly, with the variable `a` taking on successive values in the list. The output of this loop is

```
1 1
3 9
6 36
10 100
```

You could actually write the `while` example above using the built-in `range` function:

```
count = 0
for i in range(1, 101):
    if i % 2 == 0 or i % 3 == 0:
        count = count + 1
    i = i + 1
print count
```

Note that `range(a, b)` produces a list of consecutive integers starting with `a` and *ending with* `b-1`, not `b`.

Another useful built-in function, both in general and for use with loops, is `len`, which returns the length of a list or string. Here is an example:

```
x = "Vertical"
for i in range(0, len(x)):
    print x[i]
```

will print

```
V
e
r
t
i
c
a
l
```

This example shows one reason why it is useful for `range(a, b)` to go up to but not include `b`: since the list elements are indexed from 0 to the length minus 1, you can just write `range(0, len(x))` rather than needing to include a `-1` somewhere in the `range` expression.

## 12.9 Functions

Computer languages allow you not only to use built-in functions, such as `range` and `len`, but also to define and use your own functions. This is a very useful capability. For example, one function of a payroll program might be to print checks. The steps

for printing different checks will be the same other than, for example, printing different amounts and recipients. Rather than writing different code for each different check it makes sense to have a single check printing function along with some way to indicate the values, such as the check amount, that can vary.

Since you have seen some built-in functions in examples above, you already know something about function use. Specifically, to call a function you use the function name, followed by an opening parentheses, followed by one or more *arguments*, followed by a closing parentheses. The rules for function names are the same as those for variables. For example, the name can contain alphabetic characters, digits, and the underscore, but must start with an alphabetic character or an underscore.

When using a function, you need to know how many arguments it takes. For example, `len` takes a single argument that is a string or list. However, there are some functions that can have different numbers of arguments. For example, `range` can take one, two, or three arguments as the following example shows:

```
>>> print range(4)
[0, 1, 2, 3]
>>> print range(2,4)
[2, 3]
>>> print range(2, 12, 3)
[2, 5, 8, 11]
```

Function calls may be used as part of expressions, can appear more than once in an expression, and can take the results of other functions as arguments, as shown in the following example:

```
>>> a = len(range(1,101,2)) + len(range(1,101,3))
>>> print a
84
```

Defining your own function is not difficult, but there are a few rules to remember. Let's define a simple function that takes two numbers `a` and `b`, squares each (i.e., multiplies each by itself) and then adds the two results. This operation occurs often in mathematics; for example it is part of the distance formula, appears in statistics formulas, etc. Here is the example:

```
def sumsq (a, b):
    a2 = a * a
    b2 = b * b
    return a2 + b2
```

Here are the parts of the definition:

- The define line. This starts with the keyword `def`, then has the function name (in this case we have chosen `sumsq`), followed by the function *parameters* `a` and `b` that

are separated by commas and enclosed in parentheses, and the end-of-the-line colon (don't forget the colon).

(A quick terminology clarification: the terms *argument* and *parameters* are sometimes used interchangeably. We will use the convention that arguments are the values passed to the function through the function call, while parameters are the variables in the function definition. So if we execute the line `print sumsq(2,4)` then 2 and 4 are arguments that get assigned to the parameters `a` and `b`, respectively.)

- The function body, which is indented and occurs after the define line. This is just a sequence of Python statements that will be executed when the function is called.
- A `return` statement that tells what value or values to return to the calling statement. The function above returns the numerical value that is the sum of the values of `a2` and `b2`. It is possible for a function to have more than one `return` statement as long as only one is reached; for example, a function with an `if-else` statement may have a return at the end of the `if` body and another at the end of the `else` body. It is also possible for a function not to have any return statement. For instance, the function might just print a message and then end, at which point execution of the program resumes after the calling statement:

```
def printHurray(n):
    for i in range(0, n):
        print "Hurray!"
```

When our `sumsq` function is called, we must supply it with two arguments that are matched with the parameters `a` and `b`. For instance, in the statement `print sumsq(3, 5)` the argument 3 is matched with the parameter `a`, and the argument 5 is matched with `b`. The function then executes the lines in its function body, and returns the value 34 to the statement `print sumsq(3,5)`, which then prints that value.

**Problem 2:** It is possible to write `sumsq` more efficiently. See if you can write it as a two line function.

There is a special kind of function called a *method* that has its own syntax since it is closely associated with a Python structure. In our study of Python we will see a few useful methods. For example, there is an `append` method for lists, whose use is illustrated by the following example:

```
>>> treeList = ["elm", "oak", "maple"]
>>> print treeList
['elm', 'oak', 'maple']
>>> treeList.append("ash")
>>> print treeList
['elm', 'oak', 'maple', 'ash']
```

Note the syntax: the variable name is followed by a dot, followed by the method name, followed by the argument in parentheses. Here is another example that inserts `gingko` at index 2 (so as the third item) in the list. This uses the `insert` method for lists.

```
>>> treeList = ["elm", "oak", "maple"]
>>> treeList.insert(2, "gingko")
>>> print treeList
['elm', 'oak', 'gingko', 'maple']
```

The notion of *scope* is important in computer programs. Function parameters and variables defined in an indented function body are not accessible outside of the function. In computer parlance, their scope is local to the function definition. Here is an example:

```
>>> def treeAdd(newTree):
...     standardTreeList = ["oak", "maple", "elm"]
...     standardTreeList.append(newTree)
...     return standardTreeList
...
>>> print treeAdd("gingko")
['oak', 'maple', 'elm', 'gingko']
>>> print standardTreeList
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'standardTreeList' is not defined
```

Let's take a moment to examine this example. The first few lines are the definition of the `treeAdd` function. Once that is complete, the `print` line outputs the result when the function is called with the argument `"gingko"`. This works fine, it is just regular function usage. However, the next line causes an error. The problem is `standardTreeList` is defined within the function `treeAdd` and so it cannot be accessed outside that function. Another way of putting this is that the *value* of `standardTreeList` is being passed back from the function in the next-to-last line, but the variable itself (its name in particular) is not accessible outside the function.

It is actually possible to have different variables with the same name in different parts of a program as long as their scopes do not overlap. For example, a commonly used variable name such as `n` might represent a maximum possible value in many different functions in a program, in each case being local in scope to the function. This is very desirable for large projects: otherwise there would be both a profusion of awkward names, and there would need to be scrupulous care that the same names were not used for different variables.

## 12.10 Libraries

Most computer languages also include *libraries* or *modules* of items that, while sometimes used, are not used frequently. For example, Python contains a `math` module of useful data and functions. Here is a quick example:

```
import math
radius = 10.0
circumference = math.pi * radius
edge1 = 14.0
edge2 = 15.0
print math.sqrt(edge1 * edge1 + edge2 * edge2)
```

Note two important points:

- Before using data or a function in a module, you must let Python know you plan to use it. You do this by having a line starting with the `import` keyword followed by the module name.
- To use a value or function defined in a module, Python needs to know the module that defines the value, and the name of the value, separated by a period. In this case, the `math` module provides a value named `pi`, as well as a function named `sqrt`. (Remember the discussion above about method syntax.)

**Problem 3:** Rewrite the code fragment above using the `sumsq` function from earlier in this chapter.

## 12.11 Comments and Line Continuation

Comments, indicated by a pound character `#`, are used in programming to add notes for humans to read.<sup>6</sup> They are not part of the program execution, and do not alter the behavior of the program; Python ignores them. The comment can span just part of a line (the last part, after the pound sign), or an entire line. If you have a long, multiline comment start each line with a pound sign.

Another useful character in making programs more readable and understandable is the line continuation character, which in Python is the backslash `\`. The following two examples are equivalent:

```
a = 10
b = 20
print "The first value is ", a * a, " and the second is ", b * b
```

---

<sup>6</sup>Comments are often presented as being for “others” to read, but they are also useful for the code’s author, as reflected in *Eagleson’s Law*: “Any code of your own that you haven’t looked at for six or more months might as well have been written by someone else.”

and

```
a = 10
b = 20
print "The first value is ", a * a, \
      " and the second is ", b * b
```

Python ignores the backslash-newlines and combines the lines before printing them. So both the above print:

```
The first value is 100 and the second is 400
```

When a line has a long string in it, it is useful to split the string into two strings, each on different lines, rather than putting the continuation character in the middle of a string. This allows indenting the continuation of the line, which is useful to make the program readable. Compare the three print statements:

```
>>> print "It was the best of times; it was the worst \
...      of times"
It was the best of times; it was the worst      of times
>>> print "It was the best of times; it was the worst",\
... "of times."
It was the best of times; it was the worst of times.
>>> print "It was the best of times; it was the worst", \
...      "of times."
It was the best of times; it was the worst of times.
```

## 12.12 More About Lists

Being able to work with collections of data as a single entity, rather than as a number of disparate elements, is a powerful feature of programming languages. This section contains a few examples of helpful shortcuts for tasks such as initializing lists, scaling each item in a list, etc.

The example below first initializes a list to contain 31 entries with value 0, asks the user to input numbers, counts how many times certain numbers occur and updates the list accordingly, and prints out the count in two different ways. Notice the use of the following constructs:

- Initializing a long list using `range`.
- Accessing a list element using `[ ]` and `]`.
- Using a sublist: `hist[a:b]` is the sublist of items `hist[a]`, `hist[a+1]`, ..., `hist[b-1]`. Note it includes `hist[b-1]`, but not `hist[b]`.

- Getting a list length using `len`.
- Stepping through a list using `range` and `len`.

```
# Set up a list of 31 zeros.
hist = [0 for i in range(0, 31)]

# Count number of times each valid input value occurs.
p = int(raw_input("Enter a number", \
    " between 0 and 30 (-1 to exit): "))
while (p != -1):
    if (p >= 0 and p <= 30):
        # Add 1 to the count for p
        hist[p] = hist[p] + 1;
    p = int(raw_input("Enter a number", \
        " between 0 and 30 (-1 to exit): "))

# Printout 1: counts, maximum of 10 per row.
print hist[0:10]
print hist[10:20]
print hist[20:30]
print hist[30:31]

# Printout 2: 1 count per row along with index i
for i in range(0, len(hist)):
    print i, hist[i]
```

## 12.13 Random Numbers

The `random` module is another useful Python module. It can be used in simulating card games, lines in a grocery store, traffic in a transportation simulation, the spread of a disease in an epidemiological simulation, etc. Here is a short code fragment illustrating three important items for using random numbers: importing the `random` module, using the `random.randrange` function to get a random number in a discrete range of possibilities, and using the `random.random` function for generating a random floating point number.

```
import random
fourCount = 0
for i in range(0,100):
    if random.randrange(1,7) == 4:
        fourCount = fourCount + 1
print "There were ", fourCount, "4's in 100 dice rolls."
r = random.random() * 180 + 32
```

The `random.randrange(a, b)` function generates an integer between `a` and `b-1` inclusive, with each choice being equally likely. The `random.random` function generates a floating point number between 0.0 and 1.0. Note the program above first scales this range to between 0.0 and 180.0, and then adds 32 so the final range is between 32.0 and 212.0.

**Problem 4:** Write a Python expression to generate a random floating-point number in the range  $[-5, 5]$  and to store that number in the variable `d`.

## 12.14 Examples

Here are two complete examples, illustrating many of the constructs described above.

```
# Example 1
def gcd(a, b):
    # The GCD of anything and 0 is 0
    if a == 0:
        return b
    # Compute the GCD of a and b
    while b != 0:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a

# Example 2
import random
numberOfHands = int(raw_input("How many hands to deal? "))
for i in range(numberOfHands):
    fourCardHand = []
    for j in range(4):
        fourCardHand.append(random.randrange(1, 14))
    aceCount = 0
    for j in fourCardHand:
        if j == 1:
            aceCount = aceCount + 1
    print "There were ", aceCount, " aces in the hand."
print "All hands counted."
```

## 12.15 Problem Solutions

**Problem 1:** There are a number of possibilities including forgetting the colons, mistyping the variable names, and using incorrect indentation.

**Problem 2:** Here is a two line version:

```
def sumsq (a, b):  
    return (a * a) + (b * b)
```

**Problem 3:** Replace the last line with

```
print math.sqrt(sumsq(edge1, edge2))
```

**Problem 4:** Here is one possibility:

```
d = (random.random() * 10.0) - 5.0
```

## 12.16 More About Using the Python Interpreter

As explained at the start of this chapter, to use Python on the CSE lab machines, you can type `python` after opening a terminal window. This should put you in the interpreter, and you should see a `>>>` prompt. (If this doesn't work, see a TA.) At the Python prompt you can type in Python commands.

If you have a file consisting of one or more user-defined functions, you can use them in the interpreter by using the `import` command. Suppose you have a function called `coinFlip(n)` which you wrote, which takes a positive integer as input, and which is in a file `ex01.py`; then typing

```
>>> import ex01  
>>> ex01.coinFlip(100)
```

runs the function with `n = 100`.

If you have a complete program — not just a function or group of functions — and wish to run it from inside the interpreter, then you can use the `import` or `execfile` command. Here is an example of running a program `helloWorld.py` that prints “Hello, world”:

```
>>> import helloWorld  
Hello, World
```

Here is an example that uses the `execfile` command:

```
>>> execfile("helloWorld.py")  
Hello, World
```

Remember, to exit the interpreter type Control-d, i.e., hit the 'd' key while pressing down the control key at the same time.

If you have a program that you know is correct, then you can also execute it directly from the operating system command line: you can type `python programName` at the terminal window prompt to run it. Here's an example with a program `ex02.py` that prompts a user for a room length:

```
% python ex02.py
Please enter room length in inches:
...
```

You can actually do this as you develop a program: edit the program in one window, save it when you have a round of edits done, then run it in another terminal window.

## 12.17 Additional Resources

There are a vast number of online resources for Python. The beginner materials at the official Python website, <http://www.python.org>, are particularly recommended.

# Chapter 13

## Example Python Problems

### 13.1 Introduction

This chapter is a companion to the last chapter, and has two purposes: first, to provide examples of nontrivial programming-related and Python-related problems; and second, to give examples of problems that use the programming skills we would like you to be able to do for this class:

- Given Python code, be able to trace through it and tell what it returns or outputs.
- Given possibly incorrect Python code, be able to identify and correct any errors.
- Given incomplete Python code, be able to complete it.
- Given Python code that solves a problem, be able to modify it to solve a given related problem.
- Given a pseudocode description of an algorithm, be able to translate it into Python.
- Given a problem, be able to write Python code to solve the problem.

### 13.2 Problems

#### Problem 1: Tracing through a given Python program

What is printed out by each of the following code fragments?

```
# (1a)
number_of_rabbits = 2
for i in range(0, 5):
    number_of_rabbits = number_of_rabbits * 2
    print i, number_of_rabbits
```

```
# (1b)
i = 1
number_of_rabbits = 6
while i <= 16:
    if number_of_rabbits < 20:
        number_of_rabbits = number_of_rabbits * 2
    else:
        number_of_rabbits = number_of_rabbits - 10
    print i, number_of_rabbits
    i = i * 2

# (1c)
count = 0
for object in range(0, 1000):
    for triangle in range(0, 100):
        for vertex in range(0, 3):
            count = count + 1
print count
```

## Problem 2: Correcting a given Python program

The following Python program has numerous errors. Identify and correct the errors. Some errors are syntax errors, others are logical errors. So some will prevent the program from running; others will allow it to run, but will give incorrect output at times.

Here is what the program should do: it should ask the user to input 10 colors. It first gets color number 1 by asking for the red, green, and blue components of the color. Then it should check to see if all components are equal to each other. If they are, it should print a message that the color is a gray. If they are not, it should first print a message that the color is not a gray, and then compute the average  $(r + g + b) / 3$  and print it. The program should then repeat this process for a second color, then for a third one, and so on up to and including color number 10.

In doing this, examine the code below visually, and also use the Python interpreter.

Your answer should be a list of errors, with each error saying exactly what is wrong exactly where in the program.

```

for i in range(1,10):
    print "Please input the components for color number ", i
    r = int(raw_input("Please input the red component: "))
    g = int(raw_input("Please input the red component: "))
    b = int(raw_input("Please input the red component: "))
    if (r = b = g)
        print "That color is a gray"
    else
        print "That color is not gray."
        average = r + g + b / 3
        print "The average of its components is ", avg

```

### Problem 3: Correcting a given Python program

Suppose you have a stair stepper and can program various workouts. One option allows you to input successive numbers of minutes and step rates. The machine will then print out the total number of minutes, total number of steps, and average steps per minute (rounded down to the nearest integer). Here is an example with a workout that consists of 5 minutes at 100 steps/minute, followed by 5 minutes at 110 steps/minute, followed by 10 minutes at 130 steps/minute, followed by 5 minutes at 110 steps/minute, and concluded by 5 minutes at 100 steps/minute:

```

Input the number of minutes (0 to exit): 5
Input the step rate: 100
Input the next number of minutes (0 to exit): 5
Input the step rate: 110
Input the next number of minutes (0 to exit): 10
Input the step rate: 130
Input the next number of minutes (0 to exit): 5
Input the step rate: 110
Input the next number of minutes (0 to exit): 5
Input the step rate: 100
Input the next number of minutes (0 to exit): 0

Total number of minutes: 30
Total number of steps: 3400
Average step rate per minute: 113

```

Here is a listing of a version of a program for this that contains errors. Your job is to locate and correct all the errors it contains.

```

minTot = 0
stepTot = 0
min = int(raw_input("Input the number of minutes (0 to exit): "))
if min = 0:
    print "No minutes input."
else
    while min != 0:
        minTot = minTot + min
        stepRate = int(raw_input("Input the step rate: "))
        stepTot = stepTot + stepRate * min
        min = raw_input("Input the next number of minutes (0 to exit): ")
print "\nTotal number of minutes:", min
print "Total number of steps:", stepTot
# Average is rounded down.
print "Average step rate per minute: ", minTot/stepTot

```

The corrections all involve small changes; i.e., you should not have to add, delete, or modify large amounts of code. Remember, some errors might prevent the code from running; others might allow it to run, but produce an incorrect answer.

### Problem 4: Completing a Python program

*Run-length encoding* is a technique used in file compression. Suppose you have a list of the following sound volumes:

$$[4, 4, 4, 4, 4, 4, 2, 9, 9, 9, 9, 9, 5, 5, 4, 4].$$

Run length encoding replaces each run of repeated values by two numbers: the repeated value, and the number of times it occurs consecutively. For example, for the list above it would replace the first six 4's with the values 4, 6. It would replace the one 2 with 2, 1. It would replace the five 9's with 9, 5. It would replace the two 5's with 5, 2. And it would replace the two 4's with 4, 2. This results in the shorter list

$$[4, 6, 2, 1, 9, 5, 5, 2, 4, 2].$$

Your job in this part is to complete the function below that implements this process. Specifically, replace each comment with the correct Python code.

```
def RLE(myList):
    # set newList equal to the empty list
    i = 0
    # while i is less than the length of myList
    # Set currentVal equal to the element in myList with index i
    currentCount = 1
    while (i+currentCount < len(myList) and \
           myList[i+currentCount] == currentVal):
        # set currentCount equal to currentCount + 1
        # append currentVal to newList
        # append currentCount to newList
        i = i + currentCount
    # return the list newList
```

### Problem 5: Modifying a given Python program

There is a book called *10 Print* that contains a number of essays, cultural reflections, and technical explanations that all have as their starting point a one-line program run on the old Commodore 64 computer.<sup>1</sup>

Here is a more readable (and obviously longer) Python version of the *10 Print* program:

```
import random
while True:
    x = random.randrange(2)
    if x == 0:
        print "\\\"",
    else:
        print "/"",
```

The `import random` command tells the computer we'll be using a built-in random number generating routine. `while True` sets up an infinite loop (something we usually do *not* want to do). Each time through the loop `random.randrange(2)` generates a random number that is a 0 or a 1. The fifth line prints a *single* backslash character each time it is executed (since the backslash has a special role in print statements, to print one we need to precede it with a second backslash). The program generates random patterns of slashes, depending on the random numbers generated.

Here is a version of the program with the infinite loop changed to a finite `for` loop:

---

<sup>1</sup>See <http://10print.org> (accessed May 28, 2015). The full *10 Print* program is in the BASIC programming language and consists of the single line `10 PRINT CHR$(205.5+RND(1));:GOTO 10`.

```
# Finite for loop Python version of 10 Print
import random
for i in range(1000):
    x = random.randrange(2)
    if x == 0:
        print "\\ ",
    else:
        print "/ ",
```

**(Problem 5a)** Modify the finite `for` loop version so it counts the number of frontslashes it prints out, as well as (separately) the number of backslashes it prints out. Then, after completing the `for` loop, the program should report the number of frontslashes and the number of backslashes it has printed. Here is an example of what your program should print after printing all the slashes:

```
Number of frontslashes: 523
Number of backslashes: 477
```

**(Problem 5b)** Modify the finite `for` loop version so that the random number can be a 0, 1, 2, or 3. If the number is a 0 the program should print a backslash; otherwise if it is a 1 the program should print a frontslash; otherwise if it is a 2 the program should print a star (\*); and otherwise the program should print a vertical bar (|). Each `print` line should end with a comma so the program prints characters across the entire window rather than one per row.

**(Problem 5c)** Modify the finite `for` loop version so each time through the loop the program generates two random integers between 0 and 9, inclusive. If the two numbers are equal the program should print a star (\*). Otherwise if the first is greater, the program should print a backslash. Otherwise the program should print a frontslash.

## Problem 6: Modifying a Python program

Sometimes medical tests will result in a *false positive* — a positive test result even though the patient *does not* have the condition the test is checking — or a *false negative* — a negative test result even though the patient *does* have the condition. How common are false positives and false negatives when the test checks for a rare condition?

In this part you will modify an existing Python program. The program currently runs a simulation to check the frequency of false positives and false negatives. Specifically, it does the following. First it gets the following input from the user:

- `p`: the probability the patient has the disease the test is for;
- `q1`: the probability that a patient with the disease tests negative;
- `q2`: the probability that a patient who does not have the disease tests positive.

Then, for each of 800 patients, the program generates two random floating point numbers, both between 0 and 1. If the first number is  $< p$ , then the patient has the disease. In that case, if the second number is  $< q_1$ , then a false negative occurs. On the other hand, if the patient does not have the disease and the second random number is  $< q_2$ , then a false positive occurs. Any time a false positive or negative occurs the program prints out a message. The program also counts the number of false negatives and false positives (separately), and prints out the total counts right before it is done. Here is the code:

```
import random
s = int(raw_input("Please input a random seed, or 0 for no seed: "))
if s != 0:
    random.seed(s)
p = float(raw_input("Please input the probability the patient is infected: "))
q1 = float(raw_input("Please input the probability of a false negative: "))
q2 = float(raw_input("Please input the probability of a false positive: "))
falseNegCount = 0
falsePosCount = 0
for i in range(0, 800):
    x = random.random()
    y = random.random()
    if x < p:
        if y < q1:
            falseNegCount = falseNegCount + 1
            print "Test incorrectly indicated patient ", i, "is not infected"
        else:
            if y < q2:
                falsePosCount = falsePosCount + 1
                print "Test incorrectly indicated patient ", i, "is infected."
print "Number of false negatives: ", falseNegCount
print "Number of false positives: ", falsePosCount
```

Here is an example run:

```
Please input a random seed, or 0 for no seed: 42
Please input the probability the patient is infected: .05
Please input the probability of a false negative: .02
Please input the probability of a false positive: .01
Test incorrectly indicated patient 9 is infected.
Test incorrectly indicated patient 134 is infected.
Test incorrectly indicated patient 148 is infected.
Test incorrectly indicated patient 213 is infected.
Test incorrectly indicated patient 240 is infected.
Test incorrectly indicated patient 566 is infected.
Test incorrectly indicated patient 592 is not infected
Test incorrectly indicated patient 695 is infected.
Number of false negatives: 1
Number of false positives: 7
```

Your task is to make three changes in this program:

1. Modify the program so it no longer prints out the lines for individual patients. That is, it should no longer print out the lines

```
Test incorrectly indicated patient ...
```

However, the program should still print the total number of false negatives and false positives.

2. Have the user also input a number of patients. Call this  $n$ . The program should check  $n$ , rather than 800, patients. See the sample output below for what the output should look like.
3. After running the simulation the program should print the predicted number of false positives and false negatives. The formula for these are that the predicted number of false negatives is  $n \times p \times q_1$ . And the predicted number of false positives is  $n \times (1 - p) \times q_2$ . See the sample below for what the input should look like.

Here is an example of output of the modified program:

```
Please input a random seed, or 0 for no seed: 1066
Please input the number of patients tested: 3000
Please input the probability the patient is infected: .02
Please input the probability of a false negative: .05
Please input the probability of a false positive: .04
Number of false negatives: 5
Number of false positives: 111
Predicted number of false negatives: 3.0
Predicted number of false positives: 117.6
```

## Problem 7: Turning pseudocode into Python

Consider the following (partially complete) Python program for a simple game.

```
import random
def move(location):
    # This function needs to be completed

# Start of main program
s = int(raw_input("Please input a random seed, or 0 for no seed: "))
if s != 0:
    random.seed(s)
squareA = 0
squareB = 0
while squareA < 100 and squareB < 100:
    print "\nPlayer A move: "
    squareA = move(squareA)
    print "\nPlayer B move: "
    squareB = move(squareB)
if squareA >= 100 and squareB < 100:
    print "\nPlayer A won."
elif squareA < 100 and squareB >= 100:
    print "\nPlayer B won."
else:
    print "\nTie."
```

The game board consists of a path of squares numbered from 0 to 100, inclusive. Two players, Player A and Player B, both start on square 0. They alternate turns, starting with Player A. In each turn a player rolls two dice, and moves that number of squares ahead. If that would take the player past square 100, then they go to square 100. If the square they are on is evenly divisible by 11, then they go back 6 squares, else if they land on a square evenly divisible by 13, then they advance 4 squares. Regardless of what square they land on, if they roll doubles they get to roll again, and repeat this process until they do not roll doubles.

The players continue playing until a player reaches square 100. If Player B reaches square 100 first, then Player B wins. If Player A reaches square 100 first, Player B gets a turn to reach it also. In this case, if Player B does not reach it, then Player A wins; if Player B does, the game is a tie.

Your job is to complete the `move` function. Notice that function is called by the main program to move Player A, and then called again to move Player B. Each player's location is passed to the function, which then rolls the dice and follows the game rules to update the player's location.

To complete the `move` function, translate the pseudocode below into Python. Remember than in many cases pseudocode is very similar to Python, but in others there are

subtle or significant differences.

*Input:* a player's location.

*Output:* The function prints messages about the move, and returns the new location.

Function `move(location)`

```

1   set d1 = 0
2   set d2 = 0
3   while location < 100 and d1 equals d2
4       set d1 = random number between 1 and 6, inclusive
5       set d2 = random number between 1 and 6, inclusive
6       print "Roll: ", d1, "+", d2, "=", d1 + d2
7       if d1 equals d2
8           print "doubles"
9       set location = location + d1 + d2
10      if location > 100
11          set location = 100
12      print "On square", location
13      if location is divisible (with 0 remainder) by 11
14          set location = location - 6
15          print "back 6 to square", location
16      else if location is divisible (with 0 remainder) by 13
17          set location = location + 4
18          print "forward 4 to square", location
19  return location

```

## Problem 8: Writing Python code

Sometimes lists are almost, but not quite, in order. For example, due to clerical errors a mostly sorted list of employee ID numbers might have a few items out of order. Your job in this part is to write a Python program from scratch that will identify how many items in a list are out of order.

Specifically, your program should ask a user to input positive numbers, one at a time, until he or she inputs a 0. The program should store these numbers (but not the end 0) in a list in the order they are entered. When the list entry is done, the program should count each item in the list that is strictly less than the list item immediately before it. Then it should print the total number of items out of order. Here is an example:

```
Input the next list item (or enter 0 to end list): 2343
Input the next list item (or enter 0 to end list): 6382
Input the next list item (or enter 0 to end list): 6381
Input the next list item (or enter 0 to end list): 7899
Input the next list item (or enter 0 to end list): 7899
Input the next list item (or enter 0 to end list): 7905
Input the next list item (or enter 0 to end list): 7902
Input the next list item (or enter 0 to end list): 9814
Input the next list item (or enter 0 to end list): 0
There were 2 items out of order.
```

### Problem 9: Writing a Python program

Suppose that in a video game a character is positioned within a maze of rooms. The rooms are arranged in a grid. The number of rows and columns in the grid might vary if the program is run different times with different mazes, so let  $n$  be the number of rows, and  $m$  the number of columns. Each room has a one-way door leading out of it, so the program works with an  $n$ -row,  $m$ -column table, where each table entry is one of the following characters:

- ‘N’: the door leads to a room in the same column, but in the previous row (i.e., the row with index one less than the current room).
- ‘S’: the door leads to a room in the same column, but in the next row (i.e., the row with index one more than the current room).
- ‘E’: the door leads to a room in the same row but in the next column (i.e., the column with index one more than the current room).
- ‘W’: the door leads to a room in the same row, but in the previous column (i.e., the column with index one less than the current room).

If a character is positioned in a certain room (i.e., in a room in a given row and column), and they follow the one-way doors, will their path ever leave the maze? Your task is to write a function `followPath(myTable, n, m, r, c)` that has the following parameters:

- `myTable`: an  $n$ -row,  $m$ -column table, as described above.
- `n`: the number of rows in the table.
- `m`: the number of columns in the table.
- `r`: the character’s original row position. (Assume the rows of the table are labelled between 0 and  $n-1$ , inclusive.)

- `c`: the character's original column position. (Assume the columns of the table are labelled between 0 and `m-1`, inclusive.)

This function should move the character through the maze of rooms by following the one-way doors from room to room. If the character goes outside of the maze boundaries — they reach a row or column number outside the maze — then the function should print a message that the character has escaped the maze, print how many moves it took, and stop. But if the character's path leads them to a room they have visited before, then the program should recognize that, print a message that the character is trapped in the maze, print the number of moves taken, and then stop. The function does not need to return any value.

Here is an example of output using the following table for the maze

```
myMaze = [[ "S", "N", "W", "E", "W"], \
           [ "S", "N", "W", "E", "W"]]
```

(Note this is the test maze from the program, not the output (the output is below). However, your program should work not only with that maze, but with other more complicated mazes (that you make up) as well.) Here is the output:

```
Input a row: 3
Input a column: 4
Move to location row 3 and column 3
Move to location row 3 and column 4
The character is stuck in the maze.
Number of moves: 2
```

Here's another example with the same maze but a different starting position:

```
Input a row: 4
Input a column: 1
Move to location row 3 and column 1
Move to location row 2 and column 1
Move to location row 1 and column 1
Move to location row 0 and column 1
Move to location row -1 and column 1
The character has escaped the maze.
Number of moves: 5
```

## 13.3 Solutions

### Problem 1 Solution

(a)

0 4  
1 8  
2 16  
3 32  
4 64

(b)

1 12  
2 24  
4 14  
8 28  
16 18

(c)

300,000

## Problem 2 Solution

Here is a list of errors, following a listing of the program with the lines numbered.

```
1 for i in range(1,10):
2     print "Please input the components for color number ", i
3     r = int(raw_input("Please input the red component: "))
4     g = int(raw_input("Please input the red component: "))
5     b = int(raw_input("Please input the red component: "))
6     if (r = b = g)
7         print "That color is a gray"
8     else
9         print "That color is not gray."
10    average = r + g + b / 3
11    print "The average of its components is ", avg
```

Errors:

1. In line 1 the range should be `range(1,11)` (alternatively you could use `range(10)` and then replace `i` with `i+1` in line 2).
2. In lines 3 – 5 there is a missing parentheses at the end of each line.
3. In line 6 each equal sign should be replaced by a two equal signs: `r == g == b`.
4. There is a missing colon at the end of line 6.
5. There is a missing colon at the end of line 8.
6. In line 10 the average should be `(r + g + b) / 3`.
7. In line 11 `avg` should be replaced by `average` (or vice versa).

### Problem 3 Solution

```

# Changes are indicated by comments
minTot = 0
stepTot = 0
min = int(raw_input("Input the number of minutes (0 to exit): "))
# Correction 1: min = 0 changed to min == 0
if min == 0:
    print "No minutes input."
# Correction 2: else missing end colon
else:
    while min != 0:
        minTot = minTot + min
        stepRate = int(raw_input("Input the step rate: "))
        stepTot = stepTot + stepRate * min
        # Correction 3: min needs to be an int, so int (and
        # accompanying parentheses) added
        min = int(raw_input("Input the next number of minutes (0 to exit): "))
# Correction 4: min changed to minTot
print "\nTotal number of minutes:", minTot
print "Total number of steps:", stepTot
# Average is rounded down. (comment from original program)
# Correction 5: minTot/stepTot changed to stepTot/minTot
print "Average step rate per minute: ", stepTot/minTot

```

### Problem 4 Solution

```

def RLE(myList):
    newList = []
    i = 0
    while i < len(myList):
        currentVal = myList[i]
        currentCount = 1
        while (i+currentCount < len(myList) and \
               myList[i+currentCount] == currentVal):
            currentCount = currentCount + 1
        newList.append(currentVal)
        newList.append(currentCount)
        i = i + currentCount
    return newList

```

Note there is more than one possible solution. For example, there are multiple ways in Python to append a value to a list.

## Problem 5 Solution

Here are model solutions. In all parts there are alternative, but still correct, ways to modify the code.

```
# (5a)
import random
frontSlashCount = 0
backSlashCount = 0
for i in range(1000):
    x = random.randrange(2)
    if x == 0:
        print "\\ ",
        backSlashCount = backSlashCount + 1
    else:
        print "/ ",
        frontSlashCount = frontSlashCount + 1
print "Number of frontslashes: ", frontSlashCount
print "Number of backslashes: ", backSlashCount
```

```
# (5b)
import random
for i in range(1000):
    x = random.randrange(4)
    if x == 0:
        print "\\ ",
    elif x == 1:
        print "/ ",
    elif x == 2:
        print "* ",
    else:
        print "| ",
```

```
# (5c)
import random
for i in range(1000):
    x = random.randrange(10)
    y = random.randrange(10)
    if x == y:
        print "* ",
    elif x > y:
        print "\\ ",
    else:
        print "/ ",
```

## Problem 6 Solution

Here is a model solution. In some lines there are alternative, but still correct, ways to modify the code.

```
import random
s = int(raw_input("Please input a random seed, or 0 for no seed: "))
if s != 0:
    random.seed(s)
# Next line added
n = int(raw_input("Please input the number of patients tested: "))
p = float(raw_input("Please input the probability the patient is infected: "))
q1 = float(raw_input("Please input the probability of a false negative: "))
q2 = float(raw_input("Please input the probability of a false positive: "))
falseNegCount = 0
falsePosCount = 0
# Next line modified, with 800 replaced by n
for i in range(0, n):
    x = random.random()
    y = random.random()
    if x < p:
        if y < q1:
            falseNegCount = falseNegCount + 1
            # Print line originally at this location deleted
        else:
            if y < q2:
                falsePosCount = falsePosCount + 1
                # Print line originally at this location deleted
    print "Number of false negatives: ", falseNegCount
    print "Number of false positives: ", falsePosCount
# Next line added
print "Predicted number of false negatives: ", n * p * q1
# Next line added
print "Predicted number of false positives: ", n * (1-p) * q2
```

## Problem 7 Solution

Here is one possible solution:

```
import random
def move(location):
    d1 = 0
    d2 = 0
    while location < 100 and d1 == d2:
        d1 = random.randrange(1,7)
        d2 = random.randrange(1,7)
        print "Roll: ", d1, "+", d2, "=", d1+d2
        if d1 == d2:
            print "doubles"
        location = location + d1 + d2
        if location > 100:
            location = 100
        print "On square ", location
        if location % 11 == 0:
            location = location - 6
            print "back 6 to square", location
        elif location % 13 == 0:
            location = location + 4
            print "forward 4 to square", location
    return location
```

## Problem 8 Solution

Here is one possible solution:

```
myList = []
newItem = int(raw_input("Input the next list item (or enter 0 to end list): "))
while newItem != 0:
    myList.append(newItem)
    newItem = int(raw_input("Input the next list item (or enter 0 to end list): "))
count = 0
for i in range(1, len(myList)):
    if myList[i] < myList[i-1]:
        count = count + 1
print "There were ", count, "items out of order."
```

## Problem 9 Solution

Here is one possible solution:

```
def followPath(myMaze, n, m, r, c):
    # Set up table that keeps track of which locations have been visited
    visitedMaze = [[False for j in range(0,m)] for i in range(0,n)]
    visitedMaze[r][c] = True
    numMoves = 0
    # pathDone will change to True if the character revisits a previously
    # visited square or goes off any edge of the maze
    pathDone = False
    while pathDone == False:
        if myMaze[r][c] == "N":
            r = r - 1
        elif myMaze[r][c] == "S":
            r = r + 1
        elif myMaze[r][c] == "E":
            c = c + 1
        elif myMaze[r][c] == "W":
            c = c - 1
        print "Move to location row ", r, "and column ", c
        numMoves = numMoves + 1
        # Has character gone off an edge?
        if r < 0 or r >= n or c < 0 or c >= m:
            print "The character has escaped the maze."
            pathDone = True
        # Has character returned to a previously visited location?
        elif visitedMaze[r][c] == True:
            print "The character is stuck in the maze."
            pathDone = True
        # Otherwise, character has visited a new location
        else:
            visitedMaze[r][c] = True
    print "Number of moves: ", numMoves

# Main program (for testing the function)
myMaze = [[ "S", "N", "W", "E", "W"], \
           [ "S", "N", "W", "E", "W"]]
```

```
# get number of rows
n = len(myMaze)
# Get number of columns (assume all rows have the same number of columns)
m = len(myMaze[0])
r = int(raw_input("Input a row: "))
c = int(raw_input("Input a column: "))
followPath(myMaze, n, m, r, c)
```